Scheduling for Hardware-Software Partitioning in Embedded System Design

by

Daniel Wayne Engels

B.S., EECS, State University of New York at Buffalo (1992)
M.S., EECS, University of California, Berkeley (1995)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

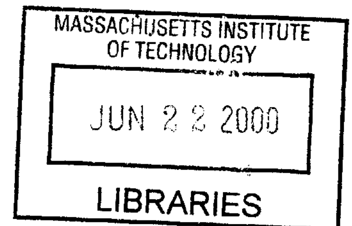
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© 2000 Massachusetts Institute of Technology. All rights reserved.

ENG



Author ....
Department of Electrical Engineering and Computer Science
17 May 2000

Certified by ....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Professor Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Scheduling for Hardware-Software Partitioning in Embedded System Design

by

Daniel Wayne Engels

Submitted to the Department of Electrical Engineering and Computer Science
on 17 May 2000, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

I present a new approach that solves the hardware-software partitioning problem for small embedded systems. Small application specific digital systems, often referred to as embedded systems, are often implemented using both hardware and software. Due to the impact that the hardware-software partition of the system functionality has on the system performance and cost, determining an optimal or near optimal hardware-software partition is essential to building a system that meets its performance criteria at minimal cost. My approach to solving the hardware-software partitioning problem is based on transforming an instance of the partitioning problem into an instance of a deterministic scheduling with rejection problem that minimizes a function of the completion times of the tasks. Although this scheduling problem is strongly $\mathcal{NP}$-hard, it has been studied extensively, and several effective solution techniques are available. I leverage these techniques to develop an efficient and effective hardware-software partitioning scheme for small embedded systems.

In addition to the new partitioning scheme, I present new complexity bounds for several variants of the scheduling problem. These complexity bounds illustrate the usefulness and futility of modifying the system characteristics to obtain a more easily solved scheduling problem formulation.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

In Memory of Lester and Margaret

Acknowledgments

The question was innocent enough. "Which will you later regret not doing?" Simple. Direct. And a revelation. That simple question held the wisdom only a father could give to his son: *What you don't do in life matters just as much as what you do.* And the answer to it changed my life. Instead of joining the real-world, I went to graduate school. I have no regrets. None. Thank you, dad. Your simple question changed my view of life and led to this dissertation.

Throughout my tenure as a gradual student, many people contributed both directly and indirectly to my growth as both a human being and a scholar. What these people have given me I could never repay. I only hope that the following adequately expresses my most sincere gratitude for their help and guidance.

I must begin by thanking my advisor, Professor Srinivas Devadas, for all that he has done for me. He took me in like a lost puppy: giving me food, shelter, and a computer to work on *before* I joined M.I.T. His support, even through unproductive times, gave me strength and allowed me to grow.

I must also thank David Karger, my de facto theory advisor and one of my readers for this dissertation. David's help and insight into theoretical arcana strongly influenced the theoretical results of this dissertation. He also instilled in me a great respect and love of the theoretical computer science world.

I thank Larry Rudolph, my second reader for this dissertation, for his refreshingly practical views on my research and our discussions on the fascinating topic of multiprocessor scheduling.

The other members of Srini's Computer Aided Automation Group made my time at M.I.T. fun and enjoyable. Silvina Hanono proved to be the best office-mate a person could have. She was always available to discuss any topic, and she gave me quiet and space when I needed it. She has proven herself to be a wonderful friend, and I look forward to many years of her continued friendship. George Hadjiyiannis, the original 'Smoky,' was always available to discuss any research topic (wink, wink, nudge, nudge), and his extensive knowledge on just about every topic never failed to

amaze me. Farzan Fallah and Sandeep Chatterjee provided hours of fun just hanging out and telling jokes (Q: What's red, black, and blue and sits in a corner? A: Dead baby. Q: What's red, black, blue, and green and sits in a corner? A: Same dead baby two weeks later.). I would also like to thank the remaining members of the group, Prabhat Jain and Todd Mills. They have helped to make my time here pleasant and enjoyable.

My friends, Gitanjali Swamy and her husband, Sanjay Sarma, have provided hours of fun, relaxation, and encouragement, as well as interesting research problems. I look forward to their continued friendship and company in the years to come.

I give special thanks to my wife, Adriana. She has suffered through years of my gradual work. In the years to come, I hope to show her that her suffering was worthwhile.

Most importantly, I thank my parents. My mother, Georgia, and my father, Keith, have always supported me in all of my endeavors. Their support and love have meant the world to me.

*Send general abuse to Stephen Edwards.

Barber ...I wanted to be a lumberjack. Leaping from tree to tree as they float down the mighty rivers of British Columbia... *(he is gradually straightening up with a visionary gleam in his eyes)* The giant redwood, the larch, the fir, the mighty scots pine. *(he tears off his barber's jacket, to reveal tartan shirt and lumberjack trousers underneath; as he speaks, the lights dim behind him and a choir of Mounties is heard faintly in the distance)* The smell of fresh-cut timber! The crash of mighty trees! *(moves to stand in front of back-drop of Canadian mountains and forests)* With my best girlie by my side... *(a frail adoring blonde, the heroin of many a mountains film, or perhaps the rebel maid, rushes to his side and looks adoringly into his eyes)* We'd sing...sing...sing.

The choir is loud by now and music as well.

Barber *(singing)* I'm a lumberjack and I'm OK,
I sleep all night and I work all day.

Lights come up to his left to reveal a choir of Mounties.

Mounties Choir He's a lumberjack and he's OK,
He sleeps all night and he works all day.

Barber I cut down trees, I eat my lunch,
I go to the lavatory.
On Wednesday I go shopping,
And have buttered scones for tea.

Mounties Choir He cuts down trees, he eats his lunch,
He goes to the lavatory.
On Wednesday he goes shopping,
And has buttered scones for tea.
He's a lumberjack and he's OK,
He sleeps all night and he works all day.

Barber I cut down trees, I skip and jump,
I like to press wild flowers.
I put on women's clothing
And hang around in bars.

Mounties Choir He cuts down trees, he skips and jumps,
He likes to press wild flowers.
He puts on women's clothing
And hangs around in bars...?

During this last verse, the choir has started to look uncomfortable, but they brighten up as they go into the chorus.

Mounties Choir He's a lumberjack and he's OK,
He sleeps all night and he works all day.

Barber I cut down trees, I wear high heels,
Suspenders and a bra.
I wish I'd been a girlie,
Just like my dear Mama.

Mounties Choir *(starting hastily as usual but tailing off as they get to the third line)*

He cuts down trees,
he wears high heels,
(spoken rather than sung) Suspenders...and a bra?...

They all mumble. Music runs down. The girl looks horrified and bursts into tears. The choir start throwing rotten fruit at him.

Girl Oh Bevis! And I thought you were so rugged.

About the Author

Daniel Wayne Engels was born in the untamed wilderness of the Dakota Territory in the long cold winter of 1970. He spent his first formative year dodging wild natives, playing with Jackrabbits, and eating lots of chocolate pudding. Having conquered the Dakota Territory by his first birthday, and eaten his store of chocolate pudding, Danny (as he was known then) set out to travel the world in search of more chocolate pudding and less snow. Danny was mildly successful in his travels despite being limited to traveling primarily in the back seat of a car. His travels brought him to pudding meccas such as Fort Hood, Texas, Fort Lewis, Washington, Fort Leavenworth, Kansas (just visiting, honest), and his favorite — he says with a nod and a wink and a *small* touch of sarcasm — Fort Drum, New York. Along the way, he found much pudding: chocolate, vanilla, and (Yuck!) tapioca,

To combat the effects large quantities of chocolate pudding (and other sugary substances) can have on the human body, Danny took up that most abusive of sports, wrestling. He quickly mastered the art of grappling with sweaty, smelly, and, later in his career, hairy men, winning several national championships as a young boy. Dan (as he was later known) continued wrestling through college. His mastery of wrestling allowed him to become an NCAA All-American in the 177 pound weight class.

Having been confined to the University at Buffalo for four long cold winters, Dan again found himself yearning to travel the world in search of more chocolate pudding and less snow. Thus, in 1992, after receiving his B.S. in Electrical Engineering and Computer Science, Dan moved West. Still being limited to traveling primarily by car, Dan stopped when he reached the snow-less hills of Berkeley, California. There, he enjoyed the nice weather, discovered that coffee suppressed his desire for pudding, and continued his studies.

Dan's stay in California would be brief and restless, partly due to the excessive consumption of coffee and partly due to the chemical imbalance called LOVE. This combination often led Dan's thoughts and body back East. And, in 1995 after receiving the M.S. in Electrical Engineering and Computer Science from the University of California, Berkeley, Dan set off on his seventh (and hopefully last) lonesome, coffee filled drive across the country.

Upon arrival Back East, Dan found himself in front of 77 Massachusetts Avenue. The temptation to further abuse his mind and pocketbook proved too great, and he quickly enrolled in the graduate program at M.I.T. After some of the most snow filled winters in Massachusetts history (and too much coffee), Dan again yearns to travel the world in search of more chocolate pudding and less snow.

Contents

1 Introduction **21**

1.1 Introduction 21

1.2 The Hardware-Software Partitioning Problem 24

1.2.1 Architecture Selection 25

1.2.2 Task Clustering 25

1.2.3 Allocation and Scheduling 26

1.2.4 Inter-Task Communication 27

1.3 Previous Partitioning Approaches 28

1.4 Introduction to Scheduling Problems 29

1.4.1 Job Characteristics 30

1.4.2 Specifying Scheduling Problems 30

1.5 Dissertation Contributions and Overview 32

2 The Scenic Specification Language **41**

2.1 Introduction 42

2.2 Processes and Signals 43

2.3 Timing Control 46

2.3.1 A Notion of Time 46

2.3.2 Timing Control Statements 47

2.4 Specifying Reactivity 48

3 Modeling a System Specification **51**

3.1 Introduction 51

3.2 Task Regions 53
3.3 Defining Tasks 58
3.4 Determining Task Characteristics 59
3.5 Using the Tasks in Scheduling Problems 62
3.5.1 Instances of Tasks 62
3.5.2 Intertask Communication 63
3.5.3 Modeling Hardware Execution Times 63
3.5.4 Wristwatch Example 64

4 The Formulation **67**
4.1 Introduction 68
4.2 The Costs of Implementation 69
4.3 Modeling Implementation Costs in a Scheduling Problem 70
4.4 Modeling Communication Constraints 71

5 Solving the Scheduling with Rejection Problem **75**
5.1 Introduction 75
5.2 The Apparent Tardiness Cost Rule 77
5.3 Inserted Idleness 79
5.4 The Scheduling Algorithm 80

6 Experimental Results **83**
6.1 Introduction 83
6.2 Wristwatch 84
6.3 Multiple Processing Elements 87

7 The Complexity of Scheduling with Rejection **97**
7.1 Introduction 98
7.2 The Total Weighted Completion Time with Rejection 101
7.2.1 Complexity of $\sum_S w_j C_j + \sum_{\bar{S}} e_j$ 101
7.2.2 Pseudo-Polynomial Time Algorithms 103
7.3 The Weighted Number of Tardy Jobs with Rejection 107

7.3.1 Complexity of $\sum_S c_j U_j + \sum_{\bar{S}} e_j$ . . . 108
7.3.2 Pseudo-Polynomial-Time Algorithm . . . 109
7.3.3 Special Cases Solvable in Polynomial Time . . . 111
7.4 The Total Weighted Tardiness with Rejection . . . 112
7.4.1 Complexity with Arbitrary Deadlines . . . 112
7.4.2 Complexity with Common Deadline . . . 117
7.5 Common Deadline and $\sum_S w_j T_j + \sum_S c_j U_j + \sum_{\bar{S}} e_j$ . . . 121
7.5.1 A Simple Pseudo-Polynomial Time Algorithm . . . 122
7.5.2 A Fully Polynomial Time Approximation Scheme . . . 125
7.5.3 Dynamic Programming on the Rejection Costs . . . 128
7.5.4 Using the FPAS . . . 129

8 The Complexity of Scheduling with Separation Constraints **133**
8.1 Introduction . . . 134
8.2 Chain Structured Tasks . . . 137
8.2.1 $\mathbf{1|chain; l_{j,k} = l|C_{max}, \sum C_j}$ . . . 138
8.2.2 $\mathbf{1|chain; pmnt; l_{j,k} = l|C_{max}, \sum C_j}$ . . . 141
8.2.3 $\mathbf{1|chain; p_j = 1; l_{j,k} \in \{0, l\}|C_{max}, \sum C_j}$ . . . 148
8.2.4 $\mathbf{1|chain; p_j \in \{1, 2\}; l_{j,k} = L\ (L \geq 2)|C_{max}, \sum C_j}$ . . . 151
8.2.5 Complexity Boundary Analysis involving Chains . . . 154
8.3 Arbitrary Precedence Structured Tasks . . . 157
8.3.1 $\mathbf{1|prec; p_j = 1; l_{j,k} = l|C_{max}, \sum C_j}$ . . . 158
8.3.2 Complexity Boundary Analysis . . . 162
8.4 Approximation Bounds for $\mathbf{P|prec; p_j = 1; l_{i,j}|C_{max}}$ . . . 164
8.4.1 Introduction . . . 164
8.4.2 List Schedules . . . 166
8.4.3 The Coffman-Graham Algorithm . . . 169

9 Conclusions **181**

A Digital Wristwatch Example **187**

List of Figures

1.1 A simple single processor, single ASIC embedded system architecture. 25

2.1 BasicWatch example Scenic process definition. 43

2.2 Wristwatch example main Scenic file. 45

3.1 Canonical task region covers of BasicWatch example process. 56

3.2 Transformation of a `wait()` node in the CFG. 57

3.3 The complete task graph for the Wristwatch example. 65

5.1 My greedy heuristic algorithm for the scheduling with rejection problem. 81

6.1 Hardware-Software partitioning results for the wristwatch example. . 86

7.1 Objective function complexity hierarchy. 98

7.2 Complexity boundary for problems $1||(\gamma + \sum_{\bar{S}} e_j)$. 100

7.3 Complexity boundary for problems $1|d_j = d|\gamma$. 100

7.4 An algorithm to solve $1||(\sum_S w_j T_j + \sum_{\bar{S}} e_j)$ using the FPAS from Section 7.5.2. 131

8.1 Complexity boundary for problems $1|chain; \beta_2; \beta_3; \beta_5|C_{\max}, \sum C_j$. . . . 135

8.2 Complexity boundary for problems $1|\beta_1; p_j = 1; \beta_5|C_{\max}, \sum C_j$. 136

8.3 Template for the proof of Theorem 8.2.1. 139

8.4 Template for the proof of Theorem 8.2.3. 143

8.5 Template for the proof of Theorem 8.2.8. 152

List of Tables

6.1 Average time to solve the Wristwatch example. 87

6.2 Prakash and Parker's examples. 89

6.3 Hou's examples. 90

6.4 Yen's large random examples. 91

6.5 MOGAC's very large random examples. 92

7.1 Scheduling with rejection complexity results. 99

8.1 Scheduling with separation constraints complexity results. 134

8.2 Complexity boundary for problems $1|chain;\beta_2;\beta_3;\beta_5|C_{\max},\sum C_j$. . . . 156

8.3 Complexity boundary for problems $P|chain;\beta_2;\beta_3;\beta_5|C_{\max},\sum w_jC_j$. . 158

8.4 Complexity boundary for problems $P|\beta_1;\beta_2;\beta_3;\beta_4;\beta_5|C_{\max},\sum C_j$. . . 163

Scheduling for Hardware-Software Partitioning in Embedded System Design

Daniel Wayne Engels

"Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning," the King said, very gravely, "and
go on 'till you come to the end: then stop."

Lewis Carroll, *Alice's Adventures in Wonderland*

Chapter 1

Introduction

"I know some new tricks,"
Said the Cat in the Hat.
"A lot of good tricks.
I will show them to you.
Your mother
will not mind at all if I do."

Dr. Seuss, *The Cat in the Hat*

This dissertation presents a novel scheduling-based approach to solving the hardware-software partitioning problem in embedded system design and examines its computational complexity. This chapter lays the foundation for understanding the approach, motivates the need for implementing embedded systems as a mix of both hardware and software, and defines the hardware-software partitioning problem. A scheduling-based approach is used to solve the partitioning problem; therefore, a brief introduction to scheduling problems and corresponding notation are described. Finally, an overview of the approach is provided, as well as a summary of the contributions of this dissertation.

1.1 Introduction

Small embedded systems, such as engine management units, dishwasher controllers, and electronic thermostats, implement dedicated, application specific functions. The

functionality of an embedded system is specified prior to its implementation, and little or no functional modification is allowed once it is installed in its operating environment. This inflexibility forces the correctness of the embedded system, both its functional correctness and its temporal correctness, to be stressed during the design process.

Many embedded systems, such as MP3 players, digital cameras, and cellular telephones, require implementations that exhibit low manufacturing cost, low power consumption, and correct timing functionality. These requirements are best satisfied by full custom hardware designs. Unfortunately, designing an entire complex embedded system as an Application Specific Integrated Circuit (ASIC) is a time consuming process that can add significant design costs to the product. Short product design cycles further deter the use of full custom hardware implementations.

By implementing some of the embedded system's functionality in software, both the design cost and the design time are reduced. A software prototype of the system functionality is often developed to evaluate the system. The software prototype allows a complete simulation of the system's behavior, permitting the designer to validate the system specification. A functionally complete and correct software prototype for a system is a functionally correct *implementation* of the system. Consequently, implementing some or all of the system functionality in software is straightforward, requiring little additional design work beyond the completion of the prototype. Thus, the use of software reduces the design time and the design cost of the electronic embedded system.

The primary disadvantage of using software to implement system functionality is that it is often much slower than a functionally correct hardware implementation. Therefore, a temporally correct implementation may require that some functionality be implemented as custom hardware.

Temporal constraints arise due to the environment in which embedded systems operate. For example, the electronic fuel injection system in an automobile engine must not only inject the proper amount of fuel, but it must inject the fuel into the engine at the correct time. Temporal, or real-time, constraints take the form of

periodicity constraints, separation constraints, and deadline constraints. Periodicity constraints require that some functionality be performed with a given frequency. For example, the frequency for sampling data in an audio system is a periodicity constraint. Separation constraints define the minimum time between the occurrence of two events. For example, the minimum time from the mixing of two chemicals until the resultant solution can be used is a separation constraint. Deadline constraints define the maximum time between the occurrence of an event and the reaction to it. For example, the maximum time to deploy an air-bag once a frontal collision is detected is a deadline constraint.

A correct system design is dependent upon the system functionality being partitioned into hardware and software components such that all temporal constraints are met in the final implementation. This *Hardware-Software Partitioning Problem* requires that, in addition to allocating system functionality to either hardware or software, the temporal behavior of the allocated functionality must be determined, either by scheduling the functionality or by other timing estimation techniques. The computational intractability of the hardware-software partitioning problem makes it impractical to manually partition complex system functionality. Therefore, automated design techniques are required to effectively search the design space for an optimal or near optimal design.

This dissertation presents a new automated approach that solves the hardware-software partitioning problem in embedded system design. The approach is based on transforming an instance of the hardware-software partitioning problem to an instance of a deterministic task scheduling with rejection problem. Solving the scheduling with rejection problem yields a partition of the system functionality and a deterministic schedule that describes the temporal behavior of the partitioned system. The scheduled tasks correspond to functionality that is to be implemented in software, and the rejected tasks correspond to functionality that is to be implemented in hardware. The deterministic schedule of the tasks provides important feedback to the designer, allowing system timing problems and other system design problems to be identified. Furthermore, the generality of this approach allows the hardware-software

partitioning problem to be solved at any level of design abstraction and at any level of functional granularity.

The remainder of this chapter builds the foundation for understanding the approach.

1.2 The Hardware-Software Partitioning Problem

The solution to the hardware-software partitioning problem specifies the implementation of all system functionality and provides some form of estimated timing behavior for the partitioned system. To find this solution, four main subproblems must be solved:

1. *Architecture Selection:* The system architecture, in terms of number of processors, number of ASICs, and communication topology, is determined.
2. *Task Clustering:* The functionality is clustered into tasks to reduce the computational complexity of the problem.
3. *Allocation:* Each task is allocated to either an ASIC or a processor.
4. *Scheduling:* The tasks are scheduled on their allocated ASIC or processor to verify temporal correctness.

These subproblems are interrelated; thus, the solution to one subproblem affects the solution to the other subproblems. The allocation and scheduling sub-problems are $\mathcal{NP}$-hard [24]; therefore, it is unlikely that a polynomial-time algorithm can be found to optimally solve the hardware-software partitioning problem.

The subproblems may be solved individually to decrease the computational complexity of the problem. I employ this approach, solving the architecture selection and task clustering subproblems individually. I then solve the allocation and scheduling subproblems concurrently.

1.2.1 Architecture Selection

The architecture of an embedded system may be a complex multiple processor, multiple ASIC design, a simple single processor, single ASIC design, or something in between. I assume the simple single processor, single ASIC architecture shown in Figure 1.1 as the chosen architecture in my scheduling-based approach. Although simple, this architecture is applicable to a wide range of embedded systems including engine control units, MP3 players, and small Web servers. And, this architecture may be implemented as a single integrated circuit, the, so called, System on a Chip (SoC). I show how to apply my approach to more complex architectures in Chapter 6.

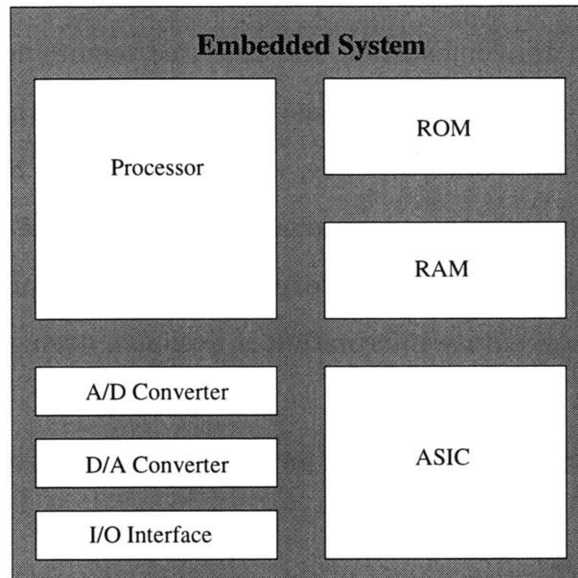


Figure 1.1: A simple single processor, single ASIC embedded system architecture.

1.2.2 Task Clustering

The functionality of a complex embedded system may contain millions of operations. Considering each operation independently while solving the hardware-software partitioning problem requires enormous quantities of computational resources. To reduce the computational complexity of the partitioning problem, the operations are clustered into a set of *tasks*. Each task corresponds to a disjoint subset of the operations

to be performed by the embedded system. The set of tasks covers all operations.

The amount of functionality modeled by a single task determines its granularity. Large granularity tasks decrease the computational complexity of the partitioning problem by reducing the number of tasks that need to be partitioned. However, large granularity tasks can lead to sub-optimal solutions since they reduce the possible number of solutions that can be determined.

The task model abstracts away the functionality of the system, but retains the pertinent characteristics of the functionality. The characteristics of a task include the area required to implement its functionality as custom hardware, the memory required to implement the task in software, and the timing behavior of both the hardware and software implementations. Additional characteristics such as power consumption for each implementation and resource requirements may be specified. The task characteristics may be determined precisely by synthesizing both hardware and software for a task. Unfortunately, synthesizing both hardware and software for all tasks is time consuming and, in general, is not possible given the short design cycles. Therefore, task characteristics are initially estimated, and the estimations may be refined when more accurate information is available during subsequent iterations of the design process.

Chapter 3 presents my methodology for determining a set of tasks from the initial system specification.

1.2.3 Allocation and Scheduling

The allocation and scheduling of the tasks is guided by system constraints and other system design issues. The system constraints include, but are not limited to, system timing constraints, hardware area constraints, software memory size constraints, and power constraints. Hard system constraints, such as a maximum application-specific hardware area constraint, cannot be violated in a feasible partition. Soft system constraints, such as a soft timing deadline, act to guide the solution by imposing a penalty when the constraint is violated. An allocation and schedule of the tasks may be feasible even if it violates soft constraints.

Chapter 4 presents my scheduling-based formulation for both the allocation and scheduling subproblems. Solving this scheduling problem simultaneously solves both the allocation and scheduling subproblems. Chapter 5 presents my algorithm for solving the scheduling problem.

1.2.4 Inter-Task Communication

An important set of constraints that I have so far neglected in my discussion of the partitioning problem are communication constraints. Communication constraints arise between tasks that must share data. Therefore, the choice of system architecture, the choice of tasks, and the allocation of the tasks determine how inter-task communication affects the solution to the partitioning problem. The effects of inter-task communication become apparent only after scheduling has been performed.

In the single processor, single ASIC architecture, communication between tasks that are both implemented in either hardware or software incur no communication penalty. However, communication between tasks that have different implementations requires additional communication time and resources. This is because the information must be transmitted either from the application-specific hardware to the software task or vice versa. The communication channels and mechanisms between hardware and software are often limited in number and bandwidth. Therefore, communication between tasks in different implementations incurs a timing penalty. The use of the communication mechanisms may be directly solved in the hardware-software partitioning problem if the target architecture, including communication mechanisms, has been determined. Otherwise, additional timing constraints may be used to model the expected communication delay, with the exact communication mechanism allocation and scheduling performed after the communication mechanisms have been defined.

I assume that the exact communication mechanisms are not known when the hardware-software partitioning problem is solved. Therefore, I model inter-task communication with separation constraints. Chapter 5 describes how my constructive algorithm adds separation constraints while it builds a solution to the allocation and scheduling subproblems.

1.3 Previous Partitioning Approaches

Hardware-software partitioning has been studied previously, yielding both exact and heuristic solution methodologies. The intractable nature of the $\mathcal{NP}$-hard hardware-software partitioning problem causes optimal approaches, such as the dynamic programming based solution of Knudsen and Madsen [36], the mixed integer linear programming formulation proposed by Schwiegershausen *et al.* [57], and the exhaustive search proposed by D'Ambrosio and Hu [14], to be useful only for small systems and simple problem formulations. Therefore, heuristics have been used to partition large systems with complex problem formulations. These heuristics typically assume a fixed set of tasks is given, and they are discussed next.

Stochastic search based algorithms, such as simulated annealing and genetic algorithms, start with a complete, but sub-optimal, solution to the partitioning problem and randomly make local changes to the solution while monitoring its cost. Algorithms using stochastic search heuristics are able to avoid being 'trapped' in locally minimal solutions, and they have been found to find optimal or near-optimal solutions [2] [29] [17] to complex problem formulations containing multiple objectives. The non-greedy, random search nature of these algorithms causes them to have long running times since they explore a large portion of the solution space. Furthermore, since their solutions are found in an essentially random manner, they often yield little insight for the designer as to how the system may be changed for the better. The implemented stochastic search approaches tend to concentrate on the allocation of tasks to either hardware or software while optimizing for the non-timing related objectives. The software tasks are scheduled as a subroutine to determine timing feasibility and solution cost.

Iterative improvement algorithms, like stochastic search based algorithms, start with a complete, but sub-optimal, solution to the partitioning problem and make local changes to the solution while monitoring its cost. The local changes are typically greedy in nature; thus, iterative improvement algorithms are prone to becoming trapped in locally optimal solutions, yielding sub-optimal solutions [30] [65] [64]. Tabu

search [21] and Kernighan and Lin-like [52] iterative improvement algorithms are two examples of iterative improvement algorithms that have been presented. The greedy nature of iterative improvement algorithms makes them have reasonable run-times; however, they suffer from the same disadvantages as the stochastic search algorithms. Their solutions often yield little insight for the designer as to how the system may be changed for the better, and the software tasks are scheduled as a subroutine to determine timing feasibility and solution cost.

Constructive algorithms, unlike stochastic search and iterative improvement algorithms, incrementally build a solution. Each decision by a constructive algorithm affects the global optimality; however, only local information is available (global information such as the impact on future decisions may only be guessed). In an attempt to gain more information, constructive algorithms often schedule the software tasks at the time that they are allocated to software. Despite a lack of global knowledge, constructive algorithms have been found to yield good solutions [15][34] with a reasonable execution time. Although many constructive algorithms perform software task scheduling while the solution is being built, the allocation of tasks to either hardware or software is usually the objective.

1.4 Introduction to Scheduling Problems

Since scheduling problems play a central role in this dissertation, let us quickly review the area of scheduling. Scheduling is concerned with the optimal allocation of resources to activities over time; therefore, they arise in all situations in which scarce resources must be allocated to activities over time. Scheduling problems have been the subject of extensive research since the early 1950's, and several surveys exist describing the results in this area [3] [20] [54] [59] [7].

A resource is typically referred to as a *machine* or *processor.* An activity is referred to as a *job* or *task.* It is assumed that a job requires at most one processor to execute at any time. Scheduling problems involve *jobs* that must be scheduled on *machines* subject to certain *constraints* to optimize some *objective function.* The goal

of the scheduling problem is to produce a *schedule* that specifies when and on which processor each task is executed such that the schedule optimizes the given objective function.

1.4.1 Job Characteristics

Each job j has several parameters associated with it. Some of these parameters characterize the job.

- The processing time p_j indicates how long it takes job j to execute to completion on a dedicated machine.
- The rejection cost e_j specifies the penalty, or cost, of not scheduling job j on a machine in the schedule.

Additional parameters specify constraints on when a job can be scheduled.

- The release time r_j indicates the first time that job j may be scheduled. Release times may never be violated in a feasible schedule.
- The deadline d_j indicates the time by which job j should be completed in the schedule. Deadlines are violated at some cost in a feasible schedule.
- The period T_j indicates the frequency at which the job becomes ready to execute. Successive instances of a job become ready to execute at times r_j, $r_j + T_j$, $r_j + 2T_j$, etc. and have respective deadlines d_j, $d_j + T_j$, $d_j + 2T_j$, etc.

Finally, cost parameters w_j and c_j are used to weight the cost of completing job j at time C_j in the schedule. The set of scheduled jobs is denoted by S, and the set of rejected jobs is denoted by $\bar{S} = N - S$, where $N = \{1, 2, \ldots, n\}$ denotes the set of jobs in the scheduling problem.

1.4.2 Specifying Scheduling Problems

There are several different scheduling problems that we examine in this dissertation. As a convenience in describing these problems, I use the $\alpha|\beta|\gamma$ notation introduced

by Graham, Lawler, Lenstra, and Rinnooy Kan [28]. In this notation, α denotes the *machine environment.* β denotes the *side constraints,* and γ denotes the *objective function.* The machine environment α is assigned one of values in the set $\{1, P\}$, denoting one machine and m identical parallel machines respectively. In both of these machine environments, job j takes processing time p_j regardless of which machine it is executed on. When the number of machines under consideration is fixed to $m > 1$ rather than being an input to the problem, the machine environment is assigned the value Pm.

The side constraints β denote any constraints on the job characteristics and on a feasible schedule. β is specified using five parameters, β_1, β_2, β_3, β_4, and β_5. $\beta_1 \in \{\circ$, chain, intree, outtree, tree, prec$\}$ denotes the precedence constraint topology. A precedence constraint between tasks i and j, $i \prec j$, requires that task i precede task j in a feasible schedule. $\beta_2 \in \{\circ$, pmtn$\}$ denotes the permissibility of preemptions in a feasible schedule. $\beta_3 \in \{\circ,\ p_j = 1,\ p_j \in \{a, b\}\}$ denotes restrictions on the task processing times. $\beta_4 \in \{\circ\ ,\ r_j\}$ denotes the presence of nonzero release times, and $\beta_5 \in \{\circ\ ,\ l_{i,j} = L = \mathcal{O}(1),\ l_{i,j} = l,\ l_{i,j} \in \{a, b\}\}$ denotes restrictions on the separation constraints between tasks. Separation constraints are only associated with precedence constraints $i \prec j$, and they denote the minimum time from the completion of task i to the beginning of task j in a feasible schedule. $l_{i,j} = L$ indicates that all separation constraints are equal in value, and the value is a fixed constant. $l_{i,j} = l$ indicates that all separation constraints are equal in value, but the value is given as an input to the problem. $l_{i,j} \in \{a, b\}$ indicates that the separation constraint values must be one of the values in the given finite set. The absence of constraints of type β_i, $1 \leq i \leq 5$, is denoted by a '$\circ$.' For clarity, the symbol '$\circ$' is not included in the $\alpha|\beta|\gamma$ problem statement.

The objective function γ denotes how the total cost of the schedule is to be calculated. The optimal schedule for a problem instance is one that minimizes the given objective function. The objective function is typically a function of the completion times, C_j, and the costs, w_j, c_j, and e_j, of the tasks. Common objective functions are the maximum completion time, or makespan, of the scheduled jobs S

$C_{\max} = \max_{j \in S}(C_j)$, the total (equivalently, average) completion time of the scheduled jobs $\sum_{j \in S} C_j$ (and its weighted version $\sum_{j \in S} w_j C_j$), the total tardiness of the scheduled jobs $\sum_{j \in S} T_j$ (and its weighted version $\sum_{j \in S} w_j T_j$), where $T_j = \max(0, C_j - d_j)$, and the total number of tardy jobs $\sum_{j \in S} U_j$ (and its weighted version $\sum_{j \in S} c_j U_j$), where U_j equals 1 if $T_j > 0$ and 0 otherwise. These objective functions may be augmented by the total rejection cost $\sum_{j \in \bar{S}} e_j$ if the scheduling problem allows jobs to be rejected, or not scheduled.

1.5 Dissertation Contributions and Overview

This dissertation is divided into two parts. Part I presents my scheduling-based approach to solving the hardware-software partitioning problem. Part II analyzes its complexity and derives several approximation algorithms for variants of the scheduling problem.

A scheduling problem requires tasks to schedule. Therefore, our discussion begins in Chapter 3 with my methodology for generating tasks from a system specification. While the methodology is general enough to be used with any specification language, it is more easily understood given a particular specification language. For this purpose, I use the Scenic specification language [25]. Scenic allows for the complete behavioral specification of a system as a set of concurrently executing, communicating processes. I describe the relevant features of the Scenic specification language in Chapter 2.

The methodology for deriving tasks from a system specification relies on the identification of *task regions* from the control-flow graph for the system specification. The notion of task regions builds upon theory developed to increase the efficiency of program analysis in parallel compilers [33]. A task region represents a region of functionality that may be implemented as a task in some scheduling problem. Since we are interested in only deterministic schedules, we are concerned with *static task regions.* Static task regions identify functional clusters in the system specification that may be scheduled and implemented in either hardware or software without violating the semantics of either the specification language or a deterministic scheduling

problem. There are $\mathcal{O}(E^2)$ static task regions, where E is the number of edges in the control-flow graph representing the system functionality.

In order to reduce the number of static task regions that must be considered, I define the notion of a *canonical static task region.* Canonical static task regions allow a process to be modeled as either a single task or as a chain of tasks. Therefore, the problem of determining a set of tasks to model a system specification is reduced to identifying the canonical static task regions within each process and choosing a level of granularity for the tasks.

After clustering the system functionality into a set of tasks, the allocation and scheduling subproblems are solved. I present my scheduling problem formulation for these two subproblems in Chapter 4. In the $\alpha|\beta|\gamma$ scheduling problem notation, my scheduling with rejection problem formulation is denoted as:

$$1|prec; r_j; l_{i,j}|(\sum_{j \in S} w_j T_j + \sum_{j \in \bar{S}} e_j).$$

This formulation is unique in that it is a pure scheduling problem that simultaneously solves both the allocation and the scheduling sub-problems of the hardware-software partitioning problem. A solution to the scheduling with rejection problem specifies the implementation of the tasks. (Rejected tasks are implemented in hardware, and scheduled tasks are implemented in software.) Furthermore, the deterministic schedule defines the temporal behavior of the partition.

The scheduling problem formulation assumes that scheduling jobs in software such that they finish at or before their respective deadlines incurs no cost. To achieve this formulation, timing constraints are modeled as release times and deadlines. The running time of a task in software corresponds directly to the processing time p_j in the scheduling problem formulation. Hardware execution times are modeled with separation constraints on transitive precedence constraints. Communication delays are modeled with separation constraints on the original precedence constraints. The costs w_j, c_j, and e_j model the tradeoff between a software implementation and a hardware implementation. Hard constraints, such as the maximum area for the application-specific hardware, cannot be modeled with this formulation; however, hard timing

constraints are modeled with infinite costs on w_j and c_j. The best possible result for this scheduling problem formulation schedules all jobs in software and has a cost of zero.

Chapter 5 describes my constructive heuristic algorithm used to solve the scheduling with rejection problem. The heuristic algorithm extends a known greedy constructive algorithm [49] to handle rejection and separation constraints. The constructive algorithm is based upon the Apparent Tardiness Cost (ATC) heuristic that uses the task weight, w_j, and processing time, p_j, to guess the cost of the task if it is not scheduled next. The task with the largest ATC is greedily scheduled next.

In Chapter 6, we examine the usefulness of my scheduling-based approach by applying it to the non-trivial digital wristwatch example. (A complete Scenic specification of the digital wristwatch is given in Appendix A.) The basic behavior of a digital wristwatch with stop watch and alarm functionality is described using nine Scenic processes. At a fine level of granularity there are thirty-seven canonical static task regions. Solutions were obtained for several system clock speeds targeting the single processor, single ASIC architecture in Figure 1.1. My scheduling-based approach was found to yield optimal solutions for all of the examined data points. All of the solutions were found in less than one second of CPU time, thereby providing fast feedback to the designer.

In addition to the digital wristwatch example, I examine several examples from the literature. All of these examples have predefined tasks, and their respective target architectures must be determined from a given set of processing elements. Using my scheduling problem formulation as the basis of a simple iterative algorithm, optimal solutions were found in less than 2.5 seconds for all of these examples.

Part II of this dissertation focuses on the complexity of scheduling problems that either allow rejection or contain separation constraints. The $\mathcal{NP}$-hard scheduling with rejection problem used in my approach is general enough to be used with all small embedded systems. However, less general problems, while only applicable to a subset of embedded systems, are of interest since they might be solvable in polynomial time or pseudopolynomial time. I identify some of these less general formulations and

characterize their complexity.

Chapter 7 probes the theoretical complexity of scheduling problems that allow rejection. The total weighted tardiness is a general objective function. Other objective functions, such as total number of tardy jobs and makespan, may be appropriate for some embedded systems. If these less general problem instances may be solved efficiently, then some embedded systems may be specified so as to take advantage of them.

In Section 7.2, we consider scheduling problems that optimize for the sum of weighted completion times plus total rejection penalty. The single machine version of this problem is denoted as $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$. If rejection is not allowed, the single machine problem is solvable in polynomial time using Smith's rule [58]: schedule the jobs in non-decreasing order of p_j/w_j. For any fixed number of parallel machines, the problem is $\mathcal{NP}$-hard [11]. In Section 7.2.1, I prove that when allowing rejection, the problem is $\mathcal{NP}$-hard, even on one machine. In Section 7.2.2, I give three pseudo-polynomial time algorithms for the problem $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$, proving that it is only weakly $\mathcal{NP}$-hard. The first algorithm runs in time $\mathcal{O}(n\sum_{j=1}^{n} e_j)$. The second algorithm runs in time $\mathcal{O}(n\sum_{j=1}^{n} w_j)$, and the third algorithm runs in time $\mathcal{O}(n\sum_{j=1}^{n} p_j)$. The second and third algorithms are joint work with Sudipta Sengupta [22].

In Section 7.3, we consider single machine scheduling problems that optimize for the sum of weighted number of tardy jobs plus total rejection penalty. The general problems $1||(\sum_S c_jU_j + \sum_{\bar{S}} e_j)$ and $1||(\sum_S U_j + \sum_{\bar{S}} e_j)$ are $\mathcal{NP}$-hard. This result follows from the $\mathcal{NP}$-hardness of the problem $1||(L_{\max} + \sum_{\bar{S}} e_j)$ [22]. (In Section 7.5, I give a pseudo-polynomial time algorithm proving them to be weakly $\mathcal{NP}$-hard.) Therefore, we consider the simpler problem where all deadlines are equal, i.e., $d_j = d$ for all jobs j. In Section 7.3.1, I prove that the problem $1|d_j = d|\sum c_jU_j$ is $\mathcal{NP}$-hard. In Section 7.3.2, I present a pseudo-polynomial time algorithm for the problem $1|d_j = d|(\sum_S c_jU_j + \sum_{\bar{S}} e_j)$, proving it to be weakly $\mathcal{NP}$-hard. And, in Section 7.3.3, I examine the special case problem $1|d_j = d; c_j \text{ agreeable}|(\sum_S c_jU_j + \sum_{\bar{S}} e_j)$ for which a simple greedy algorithm exists to solve it.

In Section 7.4, we consider single machine scheduling problems that optimize for the sum of weighted tardiness plus total rejection penalty. The general problems $1||(\sum_S w_j T_j + \sum_{\bar{S}} e_j)$ and $1||(\sum_S T_j + \sum_{\bar{S}} e_j)$ are $\mathcal{NP}$-hard. This result follows from the $\mathcal{NP}$-hardness of the problem $1||(L_{\max} + \sum_{\bar{S}} e_j)$ [22]. In Section 7.4.1, I present a pseudo-polynomial time algorithm for $1||(\sum_S T_j + \sum_{\bar{S}} e_j)$, proving it to be weakly $\mathcal{NP}$-hard. The problem $1||(\sum_S w_j T_j + \sum_{\bar{S}} e_j)$ is strongly $\mathcal{NP}$-hard; therefore, no pseudo-polynomial time algorithm exists to optimally solve it unless $\mathcal{P} = \mathcal{NP}$. In Section 7.4.2, we examine two special case problems that are solvable in polynomial time by greedy algorithms. The problem $1|d_j = d; w_j \text{ agreeable}| \sum w_j T_j$ is solved by scheduling the jobs in shortest processing time (SPT) first order. The problem $1|d_j = d; w_j \text{ disagreeable}| \sum w_j T_j$ is solved by scheduling the jobs in longest processing time (LPT) first order.

In Section 7.5, we consider single machine scheduling problems with a common deadline that optimize for the sum of weighted tardiness plus the sum of weighted number of tardy jobs plus the total rejection penalty. In Section 7.5.1, I present an $\mathcal{O}(n(\sum_{j=1}^{n} p_j - d))$ pseudo-polynomial time algorithm for the problem $1|d_j = d|(\sum_S w_j T_j + \sum_S c_j U_j + \sum_{\bar{S}} e_j)$. I modify this algorithm to develop a fully polynomial time approximation scheme (FPAS) for this problem in Section 7.5.2. In Section 7.5.3, I present a more traditional $\mathcal{O}(n^2 d \sum_{j=1}^{n} e_j)$ dynamic programming algorithm for this problem.

These scheduling with rejection complexity results indicate that by specifying the embedded system functionality such that the objective function may be something other than the total weighted tardiness or that all tasks have the same deadline, a guaranteed near optimal solution to the scheduling problem formulation can be obtained.

Chapter 8 probes the theoretical complexity of scheduling problems that allow separation constraints. Recall that separation constraints model the inter-task communication delays in my scheduling problem formulation. These communication delays are non-negligible in most systems. Therefore, excluding separation constraints can yield a solution that is far from optimal. Unfortunately, as the results in this chapter

clearly show, including separation constraints in the problem formulation causes the scheduling problem to be strongly $\mathcal{NP}$-hard.

The results in Chapter 8 consist of a set of strong $\mathcal{NP}$-hardness proofs that narrow the boundary between known polynomial time solvable problems involving separation constraints and known $\mathcal{NP}$-hard problems involving separation constraints. In Section 8.2, we consider single machine scheduling problems with chain precedence constraints and separation constraints. In Section 8.2.2, I prove that the problems $1|chain; pmtn; l_{i,j} = l|C_{\max}, \sum C_j$ are strongly $\mathcal{NP}$-hard by reducing the 3-Partition Problem to them. Using these results, I prove that the unit execution time problems $1|chain; p_j = 1; l_{i,j} \in \{0, l\}|C_{\max}, \sum C_j$ are strongly $\mathcal{NP}$-hard. In Section 8.2.4, I prove that the problems $1|chain; p_j \in \{1, 2\}; l_{i,j} = L\ (L \geq 2)|C_{\max}, \sum C_j$ are strongly $\mathcal{NP}$-hard.

In Section 8.3, we consider single machine scheduling problems with arbitrary precedence constraints and separation constraints. In Section 8.3.1, I prove that the problems $1|prec; p_j = 1; l_{i,j} = l|C_{\max}, \sum C_j$ are strongly $\mathcal{NP}$-hard.

These scheduling with separation constraints complexity results indicate that by considering the communication delays associated with inter-task communication, the problem becomes computationally intractable to solve optimally regardless of the other simplifications made to the system specification.

The dissertation concludes with a review of lessons learned and a summary of contributions and open problems.

Part I

Solving the Partitioning Problem

"...The name of the song is called *'Haddocks' Eyes'*!"

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is *called.* The name really *is 'The Aged Aged Man.'*"

"Then I ought to have said 'That's what the *song* is called'?" Alice corrected herself.

"No, you oughtn't; that's quite another thing! The song is called *'Ways and Means'*: but that's only what it is *called* you know!"

"Well, what *is* the song then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said. "The song really *is 'A-Sitting on a Gate'*: and the tune's my own invention."

Lewis Carroll, *Through the Looking Glass*

Chapter 2

The Scenic Specification Language

"Now! Now! Have no fear.
Have no Fear!" said the cat.
"My tricks are not bad,"
Said the Cat in the Hat.

Dr. Seuss, *The Cat in the Hat*

This chapter describes the details of the Scenic specification language that are relevant to my approach to solving the task clustering subproblem. My task clustering approach is general enough to be used with any specification language, but some of the details are more readily understood within the context of a specific specification language. I use the Scenic specification language for this purpose.

The Scenic specification language allows for a complete behavioral specification of a system's functionality in a C-like syntax. A complete description of the Scenic language is beyond the scope of this dissertation; therefore, only those aspects of the language relevant to the modeling of a Scenic system specification as a set of tasks are presented. The relevant language features are contained in the semantics of the language and in how the language forces a system to be specified. The syntax and implementation details of the language are largely irrelevant to how a system specification is modeled. Thus, the Scenic description presented here covers the basic semantics and structure imposed on a system specification. A complete description of the Scenic specification language and how to use it may be found in the Scenic user's manual [25].

2.1 Introduction

Specification languages are the medium in which system descriptions are written. As with natural languages, the specification language shapes not only how a design is expressed but also the very design itself. In essence, the specification language acts, in part, as a framework for ideas.

The Scenic specification language is a C++ based language that allows for the complete behavioral specification of a system. The behavior is specified as a set of *concurrently executing, communicating processes.* The execution semantics allow multiple processes to execute concurrently while the instructions within a single process are executed sequentially.

Communication between processes occurs through *signals* and *channels.* Signals provide a broadcast communication with non-blocking reads and writes of the signal value. Channels provide point to point buffered communication between processes. Reads from a channel are blocked when the buffer is empty, and writes to the channel are blocked when the buffer is full.

The execution semantics assume that all operations require zero time to execute. Therefore, time must be advanced explicitly in Scenic. In order to allow the specification of this temporal behavior, clocks are associated with processes. Clocks are used to define specific instances of time when events can occur. Events correspond to changes in the value of the inputs to and outputs from processes. Timing control statements explicitly advance time and update either the calling process's input signals and channels, output signals and channels, or both its input and output signals and channels.

The use of communicating processes provides a rigid framework for the system designer. The designer is forced to perform some degree of functional partitioning on the system in order to have multiple processes. Further functional partitioning is required within a process in order to properly position the timing control statements.

2.2 Processes and Signals

The fundamental unit for the specification of functionality in Scenic is the *process.* A process may be thought of as a sequential program that implements some behavior. Processes force the behavioral partitioning of the system functionality with all concurrent behavior specified within different processes.

Figure 2.1 illustrates how a process is defined in Scenic. A Scenic process definition is similar to a C++ class definition except that it must contain at least two methods: the constructor and the `entry()` method. The constructor defines the initial conditions for the process, and the `entry()` method defines the functionality of the process.

```
 1:  struct BasicWatch:  public sc_sync {                  // Definition of BasicWatch Process
 2:     // input ports
 3:     const sc_signal<std_ulogic>& toggleBeep;           // Toggle Hourly Beep
 4:     const sc_signal<std_ulogic>& watchTime;            // Current Watch Time
 5:     // output ports
 6:     sc_signal<std_ulogic>& watchBeep;                  // Beep if Top of Hour and Beep Set
 7:     sc_signal<std_ulogic>& newWatchTime;               // Current Watch Time Plus 1 Clock Cycle
 8:     // internal variables to this process
 9:     int beepStatus;                                    // To Beep or Not to Beep
10:
11:     // The constructor
12:     BasicWatch( sc_clock_edge& EDGE,                   // Constructor and its Parameters
13:        sc_signal<std_ulogic>& TOGGLE_BEEP,
14:        sc_signal<std_ulogic>& WATCH_BEEP,
15:        sc_signal<std_ulogic>& WATCH_TIME,
16:        sc_signal<std_ulogic>& NEW_WATCH_TIME)
17:     :  sc_sync(EDGE),
18:        toggleBeep(TOGGLE_BEEP),
19:        watchBeep(WATCH_BEEP),
20:        watchTime(WATCH_TIME),
21:        newWatchTime(NEW_WATCH_TIME)
22:     {
23:        newWatchTime.write( 0 );                        // Start the Watch at 0 Hour
24:        beepStatus = 0;                                 // Don't Beep on the Hour
25:     }
26:     void entry();                                      // Process Functionality Contained Here
27:  };
28:
29:  void BasicWatch::entry()                              // Definition of Process Functionality
30:  {
31:     int numBeeps;
32:     int time;
33:
34:     while( true ) {                                    // This Process Runs Forever
35:        if (toggleBeep.read == '1') {                   // Toggle Hourly Beep Status
36:           if (beepStatus == 0)
37:              beepStatus = 1;
38:           else
39:              beepStatus = 0;
40:        }
41:        time = IncrementWatchTime( watchTime.read );    // Update Watch Time
42:        newWatchTime.write( time );
43:        numBeeps = Beep( time, beepstatus );            // If on the Hour, then Beep
44:        watchBeep.write( numBeeps );
45:        wait();                                         // Pause Until Next Clock Edge
46:     }
47:  }
```

Figure 2.1: BasicWatch example Scenic process definition.

Notice that the `entry()` method contains a `while( true )` loop beginning at line 34. Recall that an embedded system continuously interacts with its environment. Therefore, the functionality of the embedded system is repeatedly executed. The `while( true )` loop defines the repeatedly executed functionality of the process; thus, it specifies the steady-state behavior of the process.

Communication between processes is specified with *signals* and *channels.* Signals provide a non-blocking broadcast communication mechanism. Reads from a signal do not remove its value; therefore, multiple processes may read a signal's value at the same time. Writes to a signal overwrite the current signal value with a new value, irrespective of whether or not the last value has been read. Due to the non-blocking semantics, signals always have a value associated with them. In this way, signals behave similar to a register in hardware. As the process in Figure 2.1 illustrates, Scenic requires explicit reads and writes to signals (at lines 41, 42, and 44).

While signals provide for basic communication between processes, they require detailed functional specifications when synchronous communication is required. In order to simplify the specification of synchronous communication and raise the level of abstraction for the system specification, Scenic provides a point-to-point buffered communication mechanism, the channel. Channels can only be created between one source process and one destination process. A channel contains zero or more buffers that act as a FIFO (First-In-First-Out) queue to hold communicated values. Reads from and writes to a channel are synchronous. Thus, reading from an empty channel causes the reading process to stall execution until a value is written to the channel, and writing to a full channel causes the writing process to stall until a value is read from the channel.

Processes, signals, and channels form the basic building blocks of a system specification. A system specification consists of a set of processes, each describing sequential behavior, that communicate with one another via signals and channels. Figure 2.2 shows the main scenic function declaring the processes and signals in a simple wristwatch example.

The process definitions, declarations, and their interconnections via signals and

```
1:      int scenic( int ac, char* av[]) {                  // Main Method for Declaring a Wristwatch
2:         // wristwatch button inputs
3:         sc_signal<std_ulogic> UL;                       // Upper Left Button
4:         sc_signal<std_ulogic> LL;                       // Lower Left Button
5:         sc_signal<std_ulogic> UR;                       // Upper Right Button
6:         sc_signal<std_ulogic> LR;                       // Lower Right Button
7:
8:         // internal signals
9:         sc_signal<std_ulogic> watchTime;                // Current Time
10:        sc_signal<std_ulogic> newWatchTime;             // Time After Increment
11:        sc_signal<std_ulogic> alarmTime;                // Alarm Time
12:        sc_signal<std_ulogic> stopTime;                 // Stopwatch Time
13:        sc_signal<std_ulogic> newStopTime;              // Stopwatch Time After Increment
14:        sc_signal<std_ulogic> toggleBeep;               // Toggle Hourly Chime On/Off
15:        sc_signal<std_ulogic> watchBeep;                // Chime if on Hour and Beep On
16:        sc_signal<std_ulogic> alarmBeep;                // Chime if Alarm Time and Alarm On
17:        sc_signal<std_ulogic> toggleAlarm;              // Toggle Alarm On/Off
18:        sc_signal<std_ulogic> setWatchPosition;         // Increment Position Value in Set Mode
19:        sc_signal<std_ulogic> nextWatchPosition;        // Move to Next Position in Set Mode
20:        sc_signal<std_ulogic> startStop;                // Start/Stop Stopwatch
21:        sc_signal<std_ulogic> stopReset;                // Reset Stopwatch to Zero
22:        sc_signal<std_ulogic> stopLap;                  // Display Stopwatch Lap Value
23:        sc_signal<std_ulogic> displayMode;              // Display Mode:  watch, alarm, stopwatch
24:
25:        // wristwatch functional outputs
26:        sc_signal<std_ulogic> beepStatus;
27:        sc_signal<std_ulogic> mainDisplay;
28:
29:        // declare the clock
30:        sc_clock clk(''CLOCK'', 100.0, 0.5, 0.0);
31:
32:        // declare the processes
33:        Button Btn(''Button'', clk.pos(),
34:           // inputs
35:           UL, LL, UR, LR,
36:           // outputs
37:           toggleBeep, toggleAlarm, displayMode, nextWatchPosition, setWatchPosition, startStop, stopReset, stopLap);
38:        SetWatch SWatch(''SetWatch'', clk.pos(),
39:           // inputs
40:           displayMode, nextWatchPosition, setWatchPosition, newWatchTime,
41:           // outputs
42:           watchTime);
43:        BasicWatch BWatch(''BasicWatch'', clk.pos(),
44:           // inputs
45:           toggleBeep, watchTime,
46:           // outputs
47:           watchBeep, newWatchTime);
48:        SetAlarm SetA(''SetAlarm'', clk.pos(),
49:           // inputs
50:           displayMode, nextWatchPosition, setWatchPosition,
51:           // outputs
52:           alarmTime);
53:        Alarm A(''Alarm'', clk.pos(),
54:           // inputs
55:           toggleAlarm, alarmTime, newWatchTime,
56:           // outputs
57:           alarmBeep);
58:        BasicStopWatch BStop(''BasicStopWatch'', clk.pos(),
59:           // inputs
60:           displayMode, startStop, stopReset,
61:           // outputs
62:           newStopTime);
63:        LapFilter Lap(''LapFilter'', clk.pos(),
64:           // inputs
65:           newStopTime, stopLap,
66:           // outputs
67:           stopTime);
68:        Beep B(''Beep'', clk.pos(),
69:           // inputs
70:           watchBeep, alarmBeep, newWatchTime
71:           // outputs
72:           beepStatus);
73:        Display D(''Display'', clk.pos(),
74:           // inputs
75:           displayMode, newWatchTime, stopTime, alarmTime,
76:           // outputs
77:           mainDisplay);
78:     }
```

Figure 2.2: Example of a main Scenic file declaring the signals and processes. This example declares the nine processes and their connecting signals for a wristwatch.

channels clearly describes the behavioral functionality of the system. It must also describe the temporal functionality of the system to ensure that communication and behavior occur at the appropriate times. Scenic provides *timing control statements* that may be called from a process to allow for the specification of temporal functionality that, along with the Scenic execution semantics, allow for the complete functional specification of a system.

2.3 Timing Control

The execution semantics for processes require that all processes execute concurrently and the instructions within a process execute sequentially and infinitely fast (except timing control statements described in this section). The execution semantics also require that a process be executed whenever a value on one of its input signals changes. Under these execution semantics, basic instructions require zero time to execute and processes will execute at a time instant until steady state is reached, i.e., no signal values change[1]. Therefore, time must be explicitly advanced to allow the system specification to describe the temporal functionality of the system. In particular, time must be advanced in order to specify how the system reacts to changes in the inputs from its environment. Timing control statements are used to explicitly advance time and specify temporal behavior in a Scenic system specification.

2.3.1 A Notion of Time

In order to advance time, the system specification must have a notion of time. Clocks are used to provide this notion of time. The clock for the wristwatch example in Figure 2.2 is defined at line 30. A clock is a signal with well-defined behavior. This behavior is defined to be the same as clocks in traditional hardware digital designs. A clock is a periodic square wave with a fixed period, fixed duty cycle, and defined *active* edge (either the positive edge or the negative edge). Multiple clocks may be

[1] A system specification that does not reach steady state has undefined behavior.

defined in the system specification. The active edges of the clocks in the system specification provide the time instances at which events occur. Events correspond to changes in the values of the inputs to and outputs from processes.

Clocks may be associated with a process. Processes with an associated clock are called *synchronous processes,* and processes without an associated clock are called *asynchronous processes.* The `BasicWatch` process defined in Figure 2.1 is synchronous since it has an associated clock. The key difference between synchronous and asynchronous processes is the number of times that the input signals and output signals of a process may be read and updated, respectively, at a particular time instant. In synchronous processes, the input signals and output signals are read and written, respectively, at most once per time instant. In asynchronous processes, the input signals and output signals may be read and written, respectively, more than once per time instant. The digital hardware analogy to this behavior is that the synchronous processes have registered inputs and outputs, while the asynchronous processes have neither registered inputs nor registered outputs.

A consequence of the semantics of synchronous and asynchronous processes is that synchronous processes need to be executed only once per period of their associated clock (at most one clock may be associated with a process), and asynchronous processes need to be executed whenever their input signal values change. More specifically, synchronous processes are executed once at every time instant that the active edge of their associated clock occurs. All executions of a synchronous process at a particular time instant yield the same result since the input signal values are only read once per time instant. Asynchronous processes, however, may be executed multiple times at every time instant. Since an asynchronous process reads its input signals and writes its output signals continuously, any change in input signal value necessitates the execution of the process.

2.3.2 Timing Control Statements

The semantics for synchronous processes require timing control statements that specify when to read the input signal values and when to write the output signal values.

Scenic provides a single timing control statement, the `wait()` statement, that specifies the time instant at which both the input signal values will be read and the output signal values will be written for the process containing the `wait()` statement. The `wait()` statement semantics cause the process executing the `wait()` statement to suspend execution until the next active edge of the clock associated with the process. The input signal values and the output signal values are updated on every active edge of the clock in the same manner that edge-triggered flip-flops update their outputs on every active edge of the clock.

2.4 Specifying Reactivity

In addition to the mechanisms for the specification of traditional behavioral and temporal functionality, Scenic provides mechanisms for the specification of reactive functionality. Reactive functionality allows a process to interact with the environment and other processes at a rate determined by the environment and the other processes. The zero time execution semantics of Scenic allow reactive behavior to be described within a process by simply specifying that a Boolean expression is to be evaluated immediately upon the execution of the process. The result of this expression may be used in one of two ways to provide reactive behavior.

1. If the expression evaluates to TRUE, then the specified behavior of the process is executed. Otherwise, the process terminates execution without performing any functionality.

2. If the expression evaluates to TRUE, then an 'event handler' function is executed. Otherwise, the default functionality of the process is executed.

Scenic provides two mechanisms, one for each form of reactive behavior, for the specification of reactive behavior in an easily identified manner: the `wait_until(` expression `)` and the `watching(` expression `)` constructs. In both constructs, expression must depend upon at least one input signal value (or input channel value); otherwise, once expression evaluates to FALSE, it will always evaluate to FALSE.

The `wait_until( expression )` construct specifies the first type of reactive behavior. The construct has the same functionality as the following code fragment.

```
1:        do {
2:           wait();
3:        } while ( !expression );
```

This mechanism has the effect of stalling the execution of the process until expression evaluates to TRUE.

The second type of reactive functionality is specified using the `watching( expression )` construct. Instead of stalling the execution of the process until the expression is satisfied, the process executes its default behavior until the expression evaluates to TRUE. Then, special functionality is performed. The special functionality is referred to as an 'event handler' since it is only executed when a particular event occurs. (Recall that an event is defined as a change in value of input signals or output signals.)

You find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets into the open and has other people looking at it.

A. A. Milne, *Winnie-the-Pooh*

Chapter 3

Modeling a System Specification

That is what the cat said ...
Then he fell on his head!
He came down with a bump
From up there on the ball.
And Sally and I,
We saw ALL the things fall!

Dr. Seuss, *The Cat in the Hat*

This chapter describes my solution to the task clustering subproblem of the hardware-software partitioning problem. I derive a set of periodic and sporadic real-time tasks from the system specification. The tasks abstract away the exact functionality of the system while providing an accurate representation of the system behavior and communication flow in the system. Furthermore, the tasks allow the scheduling and allocation subproblems to be solved as a scheduling with rejection problem.

3.1 Introduction

The task clustering subproblem of the hardware-software partitioning problem requires that the system specification be modeled as a set of tasks. The resulting task set comprises a system model, or unified representation, that models the system in an implementation independent manner. The task graph model allows for both the allocation and scheduling subproblems to be solved.

The critical question, therefore, is what set of tasks can completely model a system specification and be used as input to a scheduling with rejection problem formulation. As a system model, each task must correspond to a functional region, or *task region,* of the specification. No two tasks can model the same functionality, and the set of all tasks must model all of the system's functionality. Thus, the set of tasks must form an *irredundant cover* of the system's functionality. As input to a scheduling problem, each task must correspond to functionality that will be executed every time the schedule is executed. The scheduling problem assumes that each task is executed when it is scheduled, and it requires a bounded maximum amount of time before it completes. Thus, branch dependent task executions are not allowed.

Data dependencies and control dependencies between task regions cause precedence constraints to exist between their respective tasks. Scheduling problems treat all precedence constraints as if they are due to data dependencies. Therefore, if precedence constraints correspond to branching control flow, then all of the tasks corresponding to both the taken branch and the not taken branch will be executed. Thus, the solution to the scheduling problem will incorrectly model the system behavior. Furthermore, scheduling problems require that no cycles exist in the inputed task graph; otherwise, a schedule cannot be found. In order to ensure the proper modeling of the system behavior, we will consider only task regions that cause precedence constraints that do not correspond to branching control flow and do not cause cycles in the resulting task graph. Such task regions are referred to as *static task regions.*

The remainder of this chapter describes how tasks are derived from a system specification. Section 3.2 describes how task regions are identified from the Control Flow Graph (CFG) representing the system behavior. Section 3.3 describes how a set of tasks modeling the complete system is derived from the set of static task regions. Section 3.4 describes how the task characteristics are determined. Finally, Section 3.5 details how the set of tasks is used to properly formulate the scheduling with rejection problem.

3.2 Task Regions

The notion of task regions builds upon theory developed to increase the efficiency of program analysis in parallel compilers [33]. Johnson introduces the notion of single-entry single-exit regions to analyze the structure of a program and to increase the efficiency of control flow graph algorithms and data flow analysis. I use Johnson's notion of single-entry single-exit regions as the basis of my notion of task regions. The set of static task regions corresponds to a subset of Johnson's single-entry single-exit regions.

The identification of static task regions within a system specification begins by determining the set of single-entry single-exit (SESE) regions of the control flow graph representing the system functionality. The set of static task regions is a proper subset of the set of SESE regions. Before formally defining single-entry single-exit regions and task regions, a few definitions are required.

Definition 3.2.1 *A* control flow graph G *is a graph with distinguished nodes* `start` *and* `end` *such that every node occurs on some path from* `start` *to* `end`. `start` *has no predecessors, and* `end` *has no successors.*

Unlike the traditional definition of control flow graphs, my control flow graphs use explicit `switch` and `merge` nodes for standard control flow. In addition, since I am using Scenic as my specification language, explicit `wait` nodes are used for Scenic specific control flow.

Definition 3.2.2 *A node x is said to* dominate *node y in a directed graph if every path from* `start` *to y includes x. A node x is said to* postdominate *a node y if every path from y to* `end` *includes x.*

By convention, a node dominates and postdominates itself. The notions of dominance and postdominance can be extended to edges in the obvious way. Edge-based single-entry single-exit regions can now be defined as follows.

Definition 3.2.3 ([33]) *A* single-entry single-exit region *in graph G is an ordered edge pair (i, j) of distinct control flow edges i and j where*

1. *i dominates j,*

2. *j postdominates i, and*

3. *every cycle containing i also contains j and vice versa.*

i is referred to as the entry edge, and j is referred to as the exit edge of the SESE region. The first condition ensures that every path from `start` to within the SESE region passes through the region's entry edge i. The second condition ensures that every path from within the region to `end` passes through the region's exit edge j. The first two conditions are necessary but not sufficient to characterize SESE regions. Back-edges do not alter the dominance or postdominance relationships, and the first two conditions alone do not prohibit back edges from entering or exiting the SESE region. The third condition encodes two constraints: every path from within the region to a point 'above' i passes through j, and every path from a point 'below' j to within the region passes through i.

Single-entry single-exit regions capture the control structure of the system specification. This structure is used to determine logical regions of the Scenic specification which may be implemented as tasks. These *task regions* are defined based on the run-time semantics of the run-time scheduler to be used in the implementation. The scheduling with rejection problem generates static schedules where the run-time scheduler simply executes tasks according to the predefined schedule. With these run-time semantics, a static schedule cannot be efficiently generated in the presence of branching control flow. Furthermore, the scheduling with rejection problem requires that no cycles exist in the resulting task graph; otherwise, a schedule cannot be found. In the control flow graph, branching control flow and cycles occur only with both a `switch` node and a `merge` node. Therefore, we only consider task regions that either do not contain any `switch` or `merge` control operations or contain all paths from a `switch` control flow node to its corresponding `merge` node (including the `switch` and `merge` control operations). Such task regions are referred to as *static task regions.*

Definition 3.2.4 *A* static task region *in graph G is an ordered edge pair (i, j) of distinct control flow edges i and j where i and j form an SESE region and*

4. *all paths from i to j that contain a* `switch` *node also contain its corresponding* `merge` *node and vice versa, and*

5. *all paths from i to j that contain a* `merge` *node also contain its corresponding* `switch` *node and vice versa.*

The fourth and fifth conditions for static task regions require that edges i and j are not contained in either a branch of a branching control statement (such as an `if-then-else` statement) or the body of a loop.

If (i, j) and (j, k) are static task regions, then (i, k) is also a static task region. Therefore, a graph with E edges can have $\mathcal{O}(E^2)$ static task regions. To simplify the search for static task regions, for each edge e we want to find the smallest static task regions, if they exist, for which e is an entry edge or an exit edge. These are called the *canonical* static task regions associated with e.

Definition 3.2.5 *A static task region (i, j) is* canonical *provided*

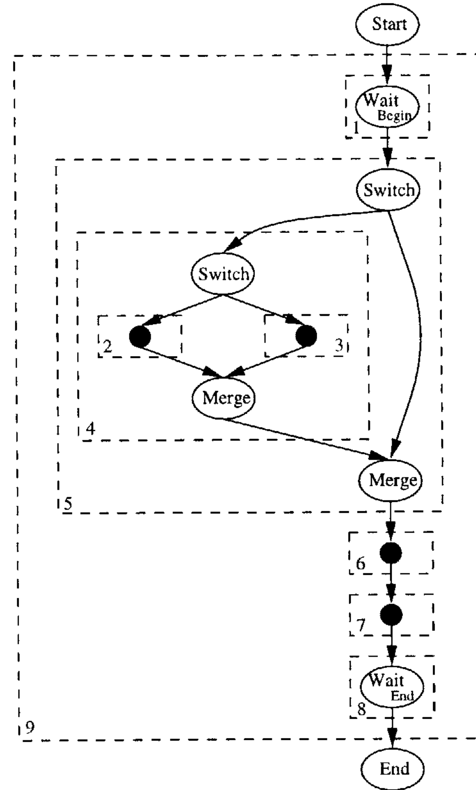
1. *j dominates j' for any static task region (i, j'), and*

2. *i postdominates i' for any static task region (i', j).*

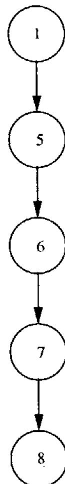
Canonical static task regions may be found in time $\mathcal{O}(E)$. In Figure 3.1(b) the canonical SESE regions are identified for the Scenic process in Figure 3.1(a). Regions 2 and 3 are disjoint. Regions 4 and 5 are nested, and regions 6 and 7 are sequentially composed. Figure 3.1(c) and Figure 3.1(d) show the sets of canonical static task regions that model the complete functionality of the Scenic process. Notice that the static task regions cannot be disjoint. They are either nested or sequentially composed as proven in the following theorems.

```
29:     void BasicWatch::entry()
30:     {
31:        int numBeeps;
32:        int time;
33:
           // while(true) loop ignored
           // while finding SESE regions
34:        while( true ) {
              // SESE 5
35:           if (toggleBeep.read == '1') {
                 // SESE 4
36:              if (beepStatus == 0)
                    // SESE 2
37:                 beepStatus = 1;
38:              else
                    // SESE 3
39:                 beepStatus = 0;
40:           }
              // SESE 6
41:           time = IncrementWatchTime( watchTime.read );
              // signal write ignored
              // while finding SESE regions
42:           newWatchTime.write( time );
              // SESE 7
43:           numBeeps = Beep( time, beepstatus );
              // signal write ignored
              // while finding SESE regions
44:           watchBeep.write( numBeeps );
              // SESE 8 and SESE 1
45:           wait();
46:        }
47:     }
```

(a) Scenic BasicWatch process



(b) Control Flow Graph with SESE regions



(c) canonical static task regions forming a task chain

(d) canonical static task region corresponding to entire process

Figure 3.1: `entry()` method of BasicWatch example (a) with SESE regions identified (b) and its two irredundant canonical task region covers (c) and (d).

Theorem 3.2.1 ([33]) *If R_1 and R_2 are two canonical SESE regions of a graph, one of the following statements applies.*

1. *R_1 and R_2 are node disjoint.*

2. *R_1 is contained within R_2 or vice versa.*

It follows from this theorem that canonical SESE regions cannot have any partial overlap. They are either disjoint, nested, or sequentially composed.

Theorem 3.2.2 *If R_1 and R_2 are two canonical static task regions of a graph, one of the following statements applies.*

1. *R_1 and R_2 are node disjoint.*

2. *R_1 is contained within R_2 or vice versa.*

Proof Follows from Theorem 3.2.1 and the definition of canonical static task regions. ■

It follows from this theorem that canonical static task regions cannot have any partial overlap. Therefore, canonical static task regions within a Scenic process are either nested or sequentially composed. Thus, the tasks created by modeling a process using multiple canonical static task regions form a chain as shown in Figure 3.1(c) and Figure 3.1(d).

Since we are defining task regions based upon *edges* in the control flow graph and we would like the Scenic `wait()` statements to delineate at least one task region, we transform a `wait()` node in the CFG into two nodes, a *begin* node and an *end* node as shown in Figure 3.2 with a directed edge from the *end* node to the *begin* node.

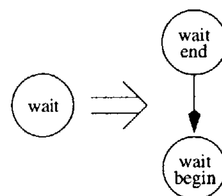


Figure 3.2: Transformation of a `wait()` node in the CFG.

This transformation ensures that at least two canonical static task regions will not contain the entire `wait` node. This property is required to ensure the correct Scenic semantics, with regard to reading and writing Scenic process input and output values respectively, in the tasks defined from these task regions (described in Section 3.3). In the remainder of this dissertation, when I refer to a `wait` node in the context of task regions, I am referring to the *begin* and *end* node transformation.

3.3 Defining Tasks

The task regions of the control-flow graph identify sequences of code that may be implemented as a single task. Based on these task regions, actual tasks may be defined in a straightforward manner. The task definition process requires the mapping of one or more task regions to a single task.

Definition 3.3.1 *A task is* properly formed *if it corresponds to a single task region or a set of sequentially composed task regions.*

Definition 3.3.2 *A properly formed task* covers *task region R if*

- *the task corresponds to R,*
- *the task corresponds to a task region that contains R,*
- *the task corresponds to a set of sequentially composed task regions that include R or a task region containing R, or*
- *the task corresponds to a set of sequentially composed task regions that include all task regions contained in R.*

Definition 3.3.3 *A set of properly formed tasks $\mathcal{T}$* covers *a control-flow graph G if every task region in G is covered by at least one task in $\mathcal{T}$.*

Definition 3.3.4 *A set of properly formed tasks $\mathcal{T}$ is an* irredundant cover *of control-flow graph G if every task region in G is covered by exactly one task in $\mathcal{T}$.*

The goal of the task definition process is to define a set of properly formed tasks that are an irredundant cover of the control-flow graph representing the Scenic specification. An irredundant cover is easily constructed from the canonical static task regions. For each set of nested regions, a level of granularity is determined, and a task corresponding to each canonical static task region at that level is created. Different levels of granularity may be determined for each set of nested regions.

The resulting irredundant cover consists of a set of task chains. The chain precedence constraints are due to the control flow within the control flow graph. Figure 3.1(c) and Figure 3.1(d) show the two sets of canonical static task regions that form irredundant covers of the control flow graph in Figure 3.1(b).

3.4 Determining Task Characteristics

After the tasks have been defined, the task characteristics must be determined. Some task characteristics are determined directly from the system specification, in particular, the release time, deadline, and period. The remaining task characteristics, such as processing time, are dependent upon the the target processor; therefore, they must be either estimated or given by the designer.

Recall that in a Scenic system specification, all Scenic processes execute independently from one another. Therefore, each Scenic process will have a unique control flow graph and, thus, a unique chain of tasks modeling its functionality. Recall, also, that a synchronous Scenic process executes with the frequency given by its associated clock. This periodic behavior is accounted for by associating a period parameter, T_j, with each task j. The task's period is used to determine the number of instances of the task that are to be scheduled as described in Section 3.5.

Task characteristics from the Scenic process are determined by analyzing the way in which the `wait` control operations are distributed throughout a chain of tasks. The trivial case is that the chain consists of a single task (see Figure 3.1(d)). This task has a one-to-one correspondence with the Scenic process; therefore, the period and deadline of the task are equal to the period of the clock associated with the Scenic

process, and the release time of the task is equal to the phase offset of the clock associated with the Scenic process. For the basic watch process of Figure 3.1, the time needs to be updated once every second. Therefore, a period of 1 second with a corresponding deadline of 1 second and release time of zero are appropriate for the task shown in Figure 3.1(d).

For chains consisting of multiple tasks, the analysis becomes more complicated. If all paths from `start` to `end` through the control-flow graph corresponding to the Scenic process encounter the same number of `wait` statements, then the Scenic process specification may be written such that no canonical task region, save the task region corresponding to the entire process, encompasses more than one `wait` control operation[1]. Let T be the period of the corresponding Scenic process. Let a be the total number of `wait` control operations encountered along all paths from `start` to `end` through the control-flow graph for the process. Consider the task that contains the ith `wait` (the *begin* portion) control operation encountered along some path. For all paths that encounter this control operation, it is the ith `wait` control operation encountered. Therefore, on the $1 + i^{\text{th}}$ clock cycle, this task will be executed. The release time of this task is iT plus the phase offset of the clock associated with the Scenic process. The period of this task is Ta. If the task also contains the *end* portion of the $1 + i^{\text{th}}$ `wait` operation, then the deadline of this task is T (the deadline of a task is specified relative to its release time); otherwise, the deadline is set to infinity.

Consider a task that does not contain the *begin* portion of a `wait` control operation. Let i be the number of `wait` control operations encountered on all paths from `start` to this task. The release time of this task can be safely set to iT without affecting the schedulability due to precedence constraints. The period of this task is Ta. If the task contains the *end* portion of the $1+i^{\text{th}}$ `wait` operation, then the deadline of this task is T; otherwise, the deadline is set to infinity.

If all paths from `start` to `end` through the control-flow graph corresponding to

[1]This may be done by unrolling loops and forcing `if` conditional statements to contain no `wait()` statements in either branch (possibly repeating the same `if` statement multiple times after multiple `wait()` statements).

the Scenic process do not encounter the same number of `wait` statements, then there will exist at least one canonical static task region that contains both the *begin* and *end* nodes corresponding to the same `wait` control operation. Such a region cannot correspond to a task since the execution semantics for all static scheduling algorithms are incapable of handling such a task while maintaining compatibility with the Scenic semantics. Therefore, processes containing such a canonical task region can only have their top-most task region, corresponding to the entire Scenic process, be used for deterministic static schedules.

The worst-case software processing time, p_j, of task j is estimated using traditional software processing time techniques [44][45]. This is a straightforward process since the functional specification in Scenic (the code contained in the `entry()` method in Figure 3.1(a)) is written in a subset of C. If a task contains multiple `wait()` statements, then the worst-case software processing time is simply the worst-case processing time among all paths beginning at a `wait()` statement and ending at the next encountered `wait()` statement.

The rejection cost e_j of task j and the task weight w_j are used to model the trade-off in the costs of being implemented in software and missing the deadline and being implemented in hardware. The rejection cost is a parameterized function of the hardware costs and interface costs. The hardware costs are estimated using traditional hardware synthesis and estimation techniques [16]. The interface logic is statically estimated based on the amount and frequency of inter-process communication with the task.

The system specification does not contain enough information on the importance of a task meeting its deadline; therefore, the designer must specify w_j. The task's weight w_j determines how tardy the task can be when it is scheduled in software before it becomes cheaper to reject the task, i.e., implement the task in hardware.

3.5 Using the Tasks in Scheduling Problems

3.5.1 Instances of Tasks

The tasks model the structure of the Scenic specification; however, they cannot be scheduled directly to determine a static deterministic schedule. Instead, *instances* of the tasks generated from the system specification are scheduled. Lawler and Martel [38] proved that statically scheduling instances of all tasks through the Least Common Multiple (LCM) of their non-zero periods yields a schedule that can be repeatedly executed with predictable results. This result eliminates the need to generate a static schedule using an infinite number of instances of each task. Therefore, we create instances of all tasks through the LCM of the non-zero periods. Tasks with periods of zero are sporadic tasks that are not associated with a clock, and techniques exist for determining the appropriate number of instances for these tasks through the LCM [47]. Note that, in the solution to the scheduling with rejection problem, some instances of a given task may be scheduled while other instances of the same task are rejected. Such solutions are feasible since we do not restrict instances of the same task to have the same implementation.

Precedence constraints exist between successive instances of a task. Therefore, instance i of task j precedes instance $i+1$ of task j. The chain structure of the tasks is maintained by requiring that instance i of the last task in a chain precede instance $i+1$ of the first task in the chain. Maintaining the chain structure ensures compliance with the Scenic semantics; otherwise, the second instance of the first task could execute before the first instance of the last task in the chain.

The release times and deadlines of the task instances are determined as follows. Let r_j be the release time of task j. Let T_j be the period of task j, and let d_j be the deadline of task j. Let r_j^i be the release time of instance i of task j, and let d_j^i be the deadline of instance i of task j. Then, $r_j^i = r_j + (i-1)T_j$ and $d_j^i = r_j + d_j + (i-1)T_j$. The deadlines of the task instances are not relative to their release times. The periods of all task instances are the same as their respective tasks.

3.5.2 Intertask Communication

The resulting set of task chains models the control flow for successive invocations of the Scenic processes. The task chains corresponding to synchronous Scenic processes may be scheduled without regard to interprocess communication. Provided that the resulting schedule meets all release times and deadlines, the schedule maintains the Scenic semantics. However, in the presence of asynchronous Scenic processes and/or the allowance of deadlines to be missed in a valid schedule (i.e., $w_j < \infty$ for some task j), interprocess communication (in the form of signals and channels) must be explicitly modeled.

I model interprocess communication simply with precedence edges. The Scenic semantics require that communication occur at the `wait` statements and at the completion of asynchronous processes. Therefore, for tasks corresponding to synchronous processes, intertask communication occurs between tasks that contain the *begin* or *end* portions of a `wait` node. For tasks corresponding to asynchronous processes, intertask communication occurs between the first task in the corresponding task chain (inputs) and the last task in the corresponding task chain (outputs).

Precedence constraints modeling the intertask communication are added such that the task containing the ith occurrence of the *begin* portion of a `wait` node in a task chain is preceded by the ith occurrence of the *end* portion of a `wait` node in the other task chains provided that the process corresponding to the task chain containing the *end* portion communicates a signal or channel to the process corresponding to the task chain containing the *begin* portion. For tasks corresponding to an entire Scenic process, the ith occurrence of an *end* portion of a `wait` node occurs in the ith instance of the task, and the ith occurrence of a *begin* portion of a `wait` node occurs in the $1 + i^{\text{th}}$ instance of the task.

3.5.3 Modeling Hardware Execution Times

After adding precedence constraints modeling intertask communication, hardware execution times and hardware-software communication costs need to be modeled. This

timing information is modeled by taking the transitive closure, G^*, of the task graph G and associating with each transitive edge a separation constraint. The separation constraint value represents the hardware execution time plus communication time if all tasks bypassed by the transitive edge are rejected. The implicit assumption is that the hardware execution time plus communication time is less than the software execution time. The resulting task graph G^* is scheduled as described in Chapter 5.

3.5.4 Wristwatch Example

The complete task graph for the digital wristwatch example is shown in Figure 3.3. The Scenic specification for the digital wristwatch is given in Appendix A. For clarity, this graph does not contain the transitive edges modeling the hardware execution times. There are nine task chains corresponding to the nine Scenic processes. The `BasicWatch`, `BasicStopWatch`, `Beep`, and `Display` processes are synchronous; all others are asynchronous. All clock periods are set to 0.01 seconds. While this requires the `BasicWatch` to count the hundredths of a second, choosing a period of 1 second for `BasicWatch` would require the LCM to be 1 second (with 100 instances of all tasks not corresponding to the `BasicWatch` process needing to be scheduled) instead of 0.01 seconds (and only 1 instance of all tasks needing to be scheduled).

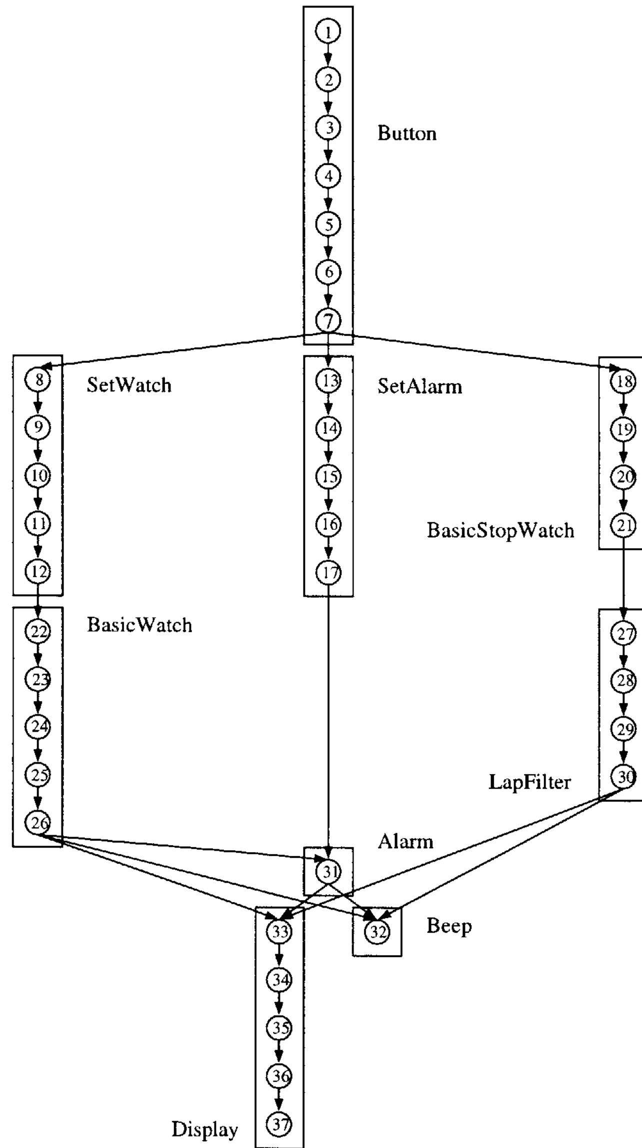


Figure 3.3: The complete task graph for the wristwatch example. The various functional modules are boxed and labeled. Precedence constraints between modules are explicit, and all synchronous tasks have a period of 0.01 seconds.

The important thing in Science is not so much to obtain new facts as to discover new ways of thinking about them.

William Lawrance Bragg, *Beyond Reductionism*

Chapter 4

The Formulation

"But I like to be here.
Oh, I like it a lot!"
Said the Cat in the Hat
To the fish in the pot.
"I will NOT go away.
I do NOT wish to go!
And so," said the Cat in the Hat,
"So
so
so . . .
I will show you
Another good game that I know!"

Dr. Seuss, *The Cat in the Hat*

I formulate and solve both the allocation and the scheduling subproblems of the hardware-software partitioning problem as a single scheduling with rejection problem. In this chapter, I show how my scheduling with rejection problem formulation properly models both the allocation and scheduling subproblems. I also show how the solution to the scheduling with rejection problem corresponds to a solution to both the allocation and scheduling subproblems for the single processor, single ASIC target architecture.

4.1 Introduction

A problem formulation *models* a real-world problem that is to be solved. In this way, the problem formulation is an abstraction of the problem. Problem formulations, in general, are used to capture specific aspects of a problem in order to reason clearly about them. Thus, problem formulations need not model all aspects and all sub-problems of a real-world problem. They may model a single sub-problem and only a few aspects of a complex real-world problem, or they may model multiple sub-problems and multiple aspects, thereby providing a more accurate model of the real-world problem.

I formulate the allocation and scheduling subproblems of the hardware-software partitioning problem as a scheduling with rejection problem. Specifically, I formulate the allocation and scheduling subproblems as the following scheduling problem:

$$1|prec; r_j; l_{i,j}|(\sum_{j\in S} w_j T_j + \sum_{j\in \bar{S}} e_j).$$

The input to this scheduling with rejection problem is a set of tasks. The tasks may have arbitrary processing times, precedence constraints, release times, deadlines, and separation constraints. The solution to this scheduling with rejection problem is a schedule of a subset of the tasks on a single processor, with the remaining tasks being rejected. The optimum schedule minimizes the sum of the sum of the weighted tardiness of the scheduled tasks and the sum of rejection costs of the rejected tasks.

A solution to this scheduling with rejection problem solves both the allocation and scheduling subproblems assuming the single processor, single ASIC target architecture of Figure 1.1. Scheduled tasks correspond to functionality that is to be implemented in software, and rejected tasks correspond to functionality that is to be implemented in hardware. The deterministic schedule provides an estimate of the timing behavior of the partitioned system.

The remainder of this chapter describes how the various costs and constraints of the hardware-software partitioning problem are formulated within the scheduling with rejection problem.

4.2 The Costs of Implementation

Let us begin by examining the cost of a single integrated circuit implementation of the single processor, single ASIC target architecture shown in Figure 1.1. The majority of the costs and constraints that must be modeled by my scheduling problem formulation are due to the implementation. Most costs are associated with the implementation of a particular task, for example, a task implemented in software has an implementation area cost due to its memory requirement. Typical constraints are either system size constraints, for example, the maximum memory used by all tasks implemented in software, or timing constraints, for example, the deadline of a particular task.

The software implementation contains both fixed and partition dependent costs. The processor on which the software executes has a fixed cost in terms of area and price. The main variable costs of the software are due to the memory requirements, the power consumption, and the timing constraints of the tasks implemented in software. The exact memory requirements of the software are a function of the tasks implemented in software. The larger the memory requirements, the greater the area required to implement the ROM and the RAM. The power consumption for the software functionality is largely determined by the speed of the clock for the microprocessor. Therefore, slower clock frequencies will reduce the power consumption but at the cost of increasing the time required to execute the software tasks. The timing constraints for the software functionality address system performance and feasibility issues. Timing constraints on the functionality of the system are violated at some cost since violations affect system feasibility and performance. Furthermore, timing constraint violations must not make the system infeasible (such violations have an infinite cost associated with them).

System functionality implemented in hardware does not have any fixed costs. The size and power requirements of the ASIC are dependent upon only those tasks implemented in hardware. Each task implemented in hardware requires a certain amount of area in the ASIC and has a certain power consumption. The more tasks implemented in hardware, the larger the ASIC area and the higher the ASIC power

consumption. Task-level timing constraints are typically not an issue in hardware since the hardware, in most instances, may be made to execute in one, or a few, clock cycles, thereby making it 'fast enough' to meet all timing constraints.

4.3 Modeling Implementation Costs in a Scheduling Problem

This section describes how a scheduling problem can model some, but not all, of the implementation costs of an embedded system. Size constraints, such as a maximum memory size, maximum ASIC area, and maximum power consumption, are not modeled by the scheduling problem formulation. Therefore, size constraints must be verified after a schedule has been generated. If a solution violates a size constraint, the designer must modify the solution or modify the system design and resolve the partitioning problem. Thus, my scheduling-based approach must be used in an iterative methodology when hard size and power constraints exist.

Since size constraints are not modeled by my scheduling problem, executing a task in software is considered to have no cost as long as its timing constraints are met. In order to capture this behavior, the software requirements of a task are modeled using the five parameters:

- processing time p_j,
- release time r_j,
- deadline d_j,
- weight w_j, and
- rejection cost e_j.

The processing time p_j indicates how long the task will take to complete if it executes without interruption on the microprocessor. The release time r_j indicates the first time at which the task may begin execution. The deadline d_j indicates the time by

which the task should be finished. The weight w_j indicates the importance of the task, and the rejection cost e_j specifies the cost of not scheduling the task.

Violating the release time of a task in some schedule incurs an infinite cost. Violating the deadline of a task incurs a cost that is a function $f(C_j, w_j)$ of the completion time C_j of the task in the schedule and the weight w_j of the task. In my scheduling problem formulation, $f(C_j, w_j) = w_j T_j$, where $T_j = \max(0, C_j - d_j)$.

Under this scheduling problem model, scheduling a task corresponds to implementing the task in software. It follows that not scheduling a task, or rejecting the task, corresponds to implementing the task in hardware. The rejection cost e_j and the weight w_j are used to determine how tardy the task may be in a schedule before it is less costly to reject the task. The rejection cost e_j does not necessarily have any relationship to the actual cost of implementing a task in hardware. Since we assume that the hardware is 'fast enough,' only the rejection cost e_j is associated with rejecting a task.

The objective of the scheduling problem is to minimize the sum of the costs incurred by each task. For my scheduling problem formulation, the objective function is $(\sum_{j \in S} w_j T_j + \sum_{j \in \bar{S}} e_j)$, where S denotes the set of scheduled tasks and $\bar{S}$ denotes the set of rejected tasks.

4.4 Modeling Communication Constraints

Communication between tasks that have different implementations must use the resources connecting the software implementation with the hardware implementation. These resources are often limited in number and bandwidth. Consequently, inter-task communication has a non-negligible effect on the temporal correctness of a partition.

The scheduling with rejection problem models inter-task communication with the use of separation constraints $l_{i,j}$. Separation constraint $l_{i,j}$ requires that task j (and all of its successors) begin execution at least $l_{i,j}$ time units after the completion of task i (and all of its predecessors) in the schedule. Separation constraints may be added to the given set of tasks as a solution is constructed. My constructive algorithm,

described in Chapter 5, adds separation constraints to the given set of tasks as it constructs a solution.

By using only separation constraints to model inter-task communication, the scheduling with rejection problem does not consider any communication resource constraints. Consequently, the separation constraints only give the expected time to transfer the data between two tasks. A more accurate schedule must be constructed after the communication resources are defined.

This page intentionally left blank[1].

[1]Except for the random text on it.

There's always an easy solution to every problem
— neat, plausible, and wrong.

H. L. Mencken

Chapter 5

Solving the Scheduling with Rejection Problem

"Now look at this trick,"
Said the cat.
"Take a look!"

Dr. Seuss, *The Cat in the Hat*

After generating a set of tasks from the system specification, the scheduling with rejection problem is solved to create a solution to the partitioning problem. One of the key advantages of formulating the partitioning problem as a scheduling problem is that scheduling problems have been studied extensively from both a practical solution perspective, as well as, a theoretical perspective. Thus, even though the scheduling with rejection problem is $\mathcal{NP}$-hard, efficient algorithms are available to find near optimal solutions to it. This chapter reviews the most effective algorithm for the weighted tardiness objective function and describes how it is extended to handle rejection and separation constraints.

5.1 Introduction

Scheduling a set of jobs so as to minimize their total weighted tardiness is a problem that has been actively investigated for more than thirty years. The continuing interest

in this problem stems from its accurate modeling of the manufacturing problem where a set of jobs must be completed by their respective deadlines, and each job incurs some penalty if it is tardy. The simplest version of this problem, $1||\sum T_j$, is weakly $\mathcal{NP}$-hard [41] [18]. Hence, a considerable number of heuristic algorithms have been proposed to solve scheduling problems involving the tardiness objective function. Most of these algorithms are based upon a greedy constructive algorithm differing only in their dispatch rules.

The simplest dispatch rules are local, i.e., the priority index calculated for a job j depends only upon that job's characteristics and the current time t. First Come-First Served (FCFS) is a commonly used benchmark rule that assigns the highest priority to the job with the earliest release time (ties broken arbitrarily). FCFS is easy to implement, but its naive approach performs poorly for most performance criteria including total tardiness.

The Earliest Due Date first (EDD) rule emphasizes job urgency by using the due date, or deadline d_j, of the job as its priority index. This rule attempts to schedule all jobs such that they complete by their respective deadlines. However, by neglecting the job processing times and weights, it may perform badly in situations where some of the jobs must be tardy. The Minimum Slack (MSLACK) rule attempts to be more intelligent by using the slack $S_j = d_j - p_j - t$ of job j as its priority index. Like EDD, MSLACK attempts to schedule all jobs such that they complete by their respective deadlines, and it may perform badly in situations where some of the jobs must be tardy. The poor performance of MSLACK arises because it does not consider job weights, and it will schedule long jobs in situations where scheduling the shortest jobs will lead to the fewest number of tardy jobs (and the smallest total tardiness), e.g., when all deadlines are equal. Ratio rules, such as Slack per Remaining Processing Time (S/RPT) [3], attempt to compensate for this tendency; when the slack becomes negative, shorter jobs will get a higher priority. The EDD, MSLACK, and S/RPT dispatch rules have been found to perform reasonably well with light processor loads, but their performance deteriorates under heavy loads [46] [61].

In contrast to the deadline oriented dispatch rules, traditional throughput oriented

dispatch rules have been used with some success. The Weighted Shortest Processing Time first (WSRPT) priority rule assigns the highest priority to the job with the largest ratio w_j/p_j. Thus, the job that incurs the largest penalty per unit of tardy processing time is scheduled first. The WSRPT rule and its unweighted variant, the Shortest Processing Time first (SPT) rule, tend to reduce congestion in the schedule by giving priority to short jobs. By emphasizing the penalty incurred by the jobs, the WSRPT rule and the SPT rule implicitly assume that the jobs are tardy. Thus, they should perform well in situations where most jobs are tardy. The SPT and WSRPT dispatch rules have been found to perform well under heavy loads with tight deadlines, but fail with light loads and loose deadlines [37] [62] [63].

Given the strengths of the above dispatch rules, an intelligent combination of the deadline oriented and throughput oriented rules should yield a dispatch rule that performs well under all system conditions. The Apparent Tardiness Cost (ATC) rule performs such a combination of the deadline oriented and throughput oriented dispatch rules, and it has been found to be superior to all of the other dispatch rules under all system conditions [37] [61] [62]. I use the ATC rule in my greedy constructive algorithm to solve the scheduling with rejection problem. Section 5.2 describes the ATC dispatch rule. Section 5.3 describes how the ATC dispatch rule is extended to allow for schedules that have inserted idleness, and Section 5.4 describes my ATC-based constructive algorithm to solve the scheduling with rejection problem.

5.2 The Apparent Tardiness Cost Rule

To overcome the deficiencies in the simple dispatch rules, more complex dispatching rules have been developed. The most successful of these heuristics is the Apparent Tardiness Cost (ATC) rule introduced by Rachamadugu and Morton [48] and extended to the multiple machine case by Vepsalainen and Morton [61]. The ATC rule is based on the structure of an optimal schedule when no precedence constraints exist between tasks. Rachamadugu [56] proved that two neighboring tasks, i and j with i

scheduled before j, in an optimal schedule obey the following priority rule:

$$\frac{w_i}{p_i}\left(1-\frac{\max\{0,(d_i-t-p_i)\}}{p_j}\right)\geq\frac{w_j}{p_j}\left(1-\frac{\max\{0,(d_j-t-p_j)\}}{p_i}\right)$$

where t is the start time of task i and the quantity on the left side of the inequality is the priority of task i. The proof of this rule is based on a pairwise interchange argument.

Rachamadugu's rule assigns a priority to a task that is equal to w_i/p_i, equivalently its WSRPT, if it is tardy and less if there is slack. Instead of trading off the slack of task i against the processing time of task j, and vice versa, a standard reference should be used. A piecewise linear look-ahead was first suggested by Carroll [12] by replacing the unknown p_j (job j is not known unless we are given an optimal schedule) in job i's priority by a factor kp, where p is the mean processing time of the unscheduled tasks and k is a look-ahead parameter related to the number of competing tardy or near-tardy tasks. However, an inverse of allowance is actually closer to the 'apparent cost' of tardiness implied by the break-even priority of tardy tasks with processing times exceeding their slack. With this in mind, the ATC dispatching rule is defined as

$$ATC_j(t)=\frac{w_j}{p_j}\exp\left(-\frac{\max\{0,(d_j-t-p_j)\}}{kp}\right),$$

where p is the mean processing time of the tasks yet to be scheduled and k is the look-ahead parameter. Intuitively, the exponential look-ahead works by ensuring timely completion of short tasks (with a steep increase of priority close to its deadline), and by extending the look-ahead far enough to prevent long tardy tasks from overshadowing clusters of shorter tasks. The look-ahead parameter can be adjusted based on the expected number of competing tasks to reduce weighted tardiness costs during high processor load. Experiments have found that a reasonable range of values for k is $1.5 \leq k \leq 4.5$ with $k = 2$ yielding good results over a wide range of load conditions [61].

Empirical experiments have found that the ATC rule yields close to optimal schedules for single machine schedules [48] and outperforms all other dispatch rules for multiple machine schedules [61]. Additionally, the ATC dispatch rule has been found to

be robust in the presence of errors in the estimated processing times of the tasks [60]. The robustness of the ATC dispatch rule in the presence of errors in processing time estimates is essential for its use in solving our scheduling with rejection problem formulation.

5.3 Inserted Idleness

Simply using the ATC dispatching rule yields a non-preemptive schedule without any inserted idle time. However, in the presence of release times, or, similarly, separation constraints, allowing inserted idle time can yield better schedules [37] with minimal additional computational expense. Morton and Ramnath [49] showed that for all problem instances and for any regular objective function, including the (weighted) tardiness objective function, there exists an optimal schedule such that no job is scheduled next on a given machine unless its release time is at most the current time plus the processing time of the shortest job that was released by the current time. Based on this fact, they proposed a modification of the ATC rule for the single machine problem. The priorities of the jobs are multiplied by a penalty proportional to the inserted idleness caused by scheduling that job next. In this way, the set of candidate jobs to be scheduled next is extended to include jobs that will arrive in the near future.

The priorities of the yet to be released jobs are reduced proportional to the idleness that would be incurred by scheduling them next. The proportionality multiplier α may be a constant, or it may be variable to allow it to increase linearly with the machine utilization as suggested by Morton and Ramnath [49]. The ATC rule that allows inserted idle time is then defined as

$$ATC_j(t)' = ATC_j(t)\left(1.0 - \alpha\frac{\max\{0, (r_j - t)\}}{p_{\min}}\right),$$

where $p_{\min}$ is the processing time of the shortest job that is ready at time t. This new ATC rule degrades the original ATC priority by a term proportional to the induced idleness as a fraction of the minimum of the processing times of the waiting jobs. If

the reduced priority of a yet to be released task is greater than all other task priorities, then the machine is kept idle until this job is released. I use this dispatching rule with a constant proportionality multiplier to allow for inserted idleness in the algorithm described in the following section.

5.4 The Scheduling Algorithm

My greedy constructive algorithm to generate a solution to the scheduling with rejection problem is simply stated as follows. At each time t that the processor becomes free, compute $ATC_j(t)'$ for all tasks j that are ready to execute at time t or become ready to execute during the interval $t + p_{\min}$, where $p_{\min}$ is the minimum processing time of the tasks that are ready to execute at time t. Let i be the task with the largest computed ATC. Let $z_i = w_i T_i$ be the weighted tardiness of task i if it is scheduled as soon as possible at or after time t. If $z_i \geq e_i$, then reject task i and repeat the task selection process at time t; otherwise, schedule task i as soon as possible at or after time t. Let the completion time of task i be the new current time t. Repeat until all jobs have been either scheduled or rejected. This algorithm runs in time $\mathcal{O}(n^2)$.

```
1:    𝒯 ⟵ set of tasks to be scheduled
2:    S ⟵ ∅                                   // set of scheduled tasks
3:    S̄ ⟵ ∅                                   // set of rejected tasks
4:    t ⟵ min_{j∈𝒯} r_j                       // start scheduling at earliest release time
5:
6:    while( 𝒯 ≠ ∅ ) {
7:       ℛ ⟵ set of ready tasks with r_j ≤ t and j ∈ 𝒯
8:       p_min ⟵ min_{j∈ℛ} p_j
9:       ℛ' ⟵ set of ready tasks with r_j ≤ t + p_min and j ∈ 𝒯
10:      i ⟵ max_{j∈ℛ'} ATC_j(t)'
11:      T_i ⟵ max(t, r_i) + p_i − d_i
12:      z_i ⟵ w_i T_i
13:      if ( z_i < e_i ) {
14:         schedule i beginning at time max(t, r_i)
15:         S ⟵ S + i
16:         𝒯 ⟵ 𝒯 − i
17:         t ⟵ max(t, r_i) + p_i
18:      } else {
19:         reject i
20:         S̄ ⟵ S̄ + i
21:         𝒯 ⟵ 𝒯 − i
22:      }
23:   }
24:
25:   return S and S̄
```

Figure 5.1: My greedy heuristic algorithm for the scheduling with rejection problem.

Pooh looked at his two paws. He knew that one of them was the right, and he knew that when you had decided which one of them was the right, then the other one was the left, but he never could remember how to begin.

A. A. Milne, *The House at Pooh Corner*

Chapter 6

Experimental Results

"I call this game FUN-IN-A-BOX,"
Said the cat.
"In this box are two things
I will show to you now.
You will like these two things,"
Said the cat with a bow.

Dr. Seuss, *The Cat in the Hat*

We have seen how to formulate the allocation and scheduling subproblems of the hardware-software partitioning problem as a scheduling problem, how to generate tasks for this problem from a system specification, and how to solve the scheduling problem formulation. Now, we examine how well this scheduling-based approach performs. In this chapter, I apply my scheduling-based partitioning approach to a simple digital wristwatch example, as well as several examples from the literature and analyze the results.

6.1 Introduction

In order to evaluate my scheduling-based approach to solving the hardware-software partitioning problem, I applied it to multiple variants of a simple digital wristwatch example. The wristwatch has the basic functionality found on most digital watches today. It displays the time in increments of one second. It has a stopwatch that is

capable of reporting the time to the hundredth of a second, and it has a daily alarm. I specified the wristwatch functionality in Scenic using nine (9) processes.

Hardware-software partitioning was performed on the wristwatch functionality at two levels of functional granularity: coarse grain (each process corresponds to a task) and fine grain (canonical static task regions within each process correspond to tasks). Partitioning was performed at multiple processor speeds targeting the single-processor, single-ASIC architecture using my scheduling-based partitioning approach. The partitioning results found that the complex display functionality should be implemented in hardware for slow clock speeds. However, at the relatively slow clock speed of 9kHz, all of the functionality could be implemented in software without violating any timing constraints. Section 6.2 presents the detailed results and analysis for the wristwatch example.

While the wristwatch example illustrates the usefulness of my approach, it does not yield any results that may be compared with previously reported approaches. In order to compare my approach to previous approaches, I applied my scheduling problem formulation for the allocation and scheduling subproblems to several examples from the literature. These examples have predefined tasks and do not assume a fixed single-processor single-ASIC target architecture. Instead, a set of Processing Elements (PEs) is given, and a subset of these elements must be chosen as the target architecture. In Section 6.3, I give a simple algorithm to solve this problem by repeatedly applying the scheduling with rejection problem formulation. My results compare favorably with those presented in the literature, finding the optimal solution for all examples in orders of magnitude less running time.

6.2 Wristwatch

The digital wristwatch example is a simplified version of the digital wristwatch commonly sold today. The basic functionality of the example is the same as that of the common digital wristwatch. The main differences occur in the extra features. For example, the wristwatch example only displays the time in a 24-hour clock mode.

There is no 12-hour AM/PM time display mode. The date and day of the week functionality also are not specified in the example wristwatch. These omissions were made to simplify the example specification, and they do not affect the validity of the analysis performed on the example.

The digital wristwatch performs three main types of functionality: time keeper, stopwatch, and daily alarm. The time keeper, or watch, functionality maintains the time for display in hours, minutes, and seconds. A chime beeps every full hour, and the time is set by executing an appropriate setting sequence. The stopwatch records the minutes, seconds, and hundredths of a second from the time it is started. A lap time measurement is provided. The chime sounds when the stopwatch is started and when it is stopped. The daily alarm sounds the chime for thirty seconds when the alarm time occurs and the alarm is enabled. The alarm time is set by executing an appropriate setting sequence.

The wristwatch has five display modes that correspond to the five different modes of operation: the watch mode, the set-watch mode, the stopwatch mode, the alarm mode, and the set-alarm mode.

The user controls the wristwatch by depressing four buttons. The upper left button is used for entering and exiting the set-watch mode and the set-alarm mode. The lower left button is used for cycling between watch mode, stopwatch mode, and alarm mode. It is also used to change the position being set when in set-watch and set-alarm modes. The upper right button is used as the `LAP` button in stopwatch mode. The lower right button applies a setting in set-watch and set-alarm modes and is the start/stop button in stopwatch mode.

The system functionality of the wristwatch is described in Scenic. A total of nine (9) Scenic processes are used to describe the behavior of the digital wristwatch. These processes contain a total of thirty-seven (37) canonical static task regions as shown in Figure 3.3.

The wristwatch is not a computationally intensive system with most of the computational complexity resulting from the watch display. Due to this lack of computational requirements, I chose the PIC16F84 microcontroller as the target processor with

the hardware tasks (the rejected tasks in my scheduling formulation) implemented as a co-processor. Since my approach is not integrated into a complete codesign system, I cannot provide the final system costs as a result of high-level synthesis.

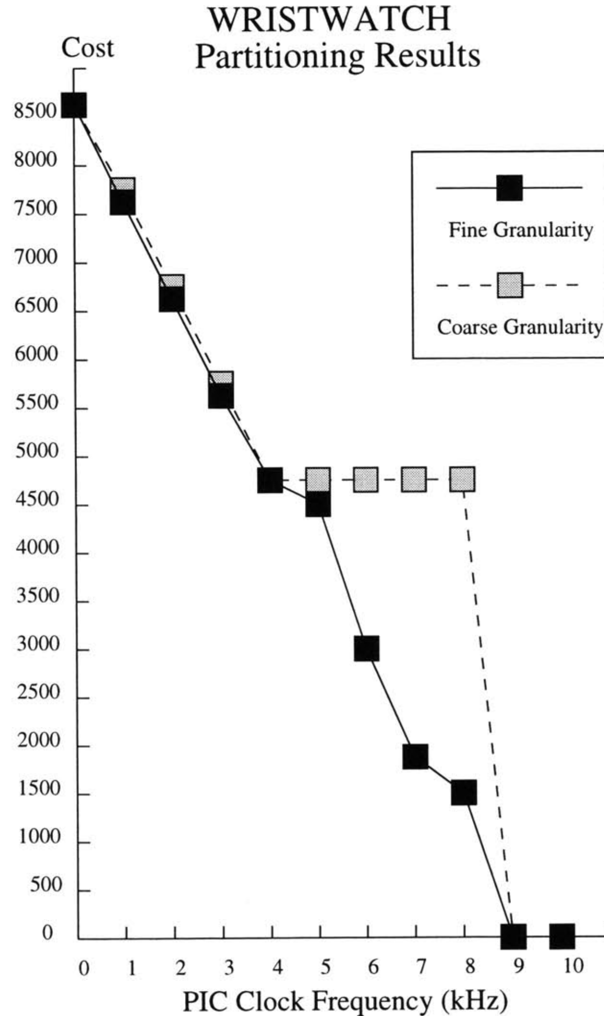


Figure 6.1: Hardware-Software partitioning results for the wristwatch example. The fine granularity partitioned 37 tasks, and the coarse granularity partitioned 9 tasks. When the processor has a low clock frequency, more functionality must be implemented in hardware with a high cost. Conversely, when the processor has a high clock frequency, all tasks meet their deadlines in software, and there is no requirement for hardware.

Figure 6.1 shows the total cost of the partition found by my approach as a function of the clock frequency of the PIC microprocessor. My approach is implemented in Java, and all examples required less than 1 second of processing time running on a 300MHz Pentium II processor. Table 6.1 shows the average time required to determine

Wristwatch		
Task Granularity	No. of Tasks	Average Time (s)
Coarse	9	0.003
Fine	35	0.013

Table 6.1: Average time to determine the data points for the Wristwatch example data points in Figure 6.1.

a datapoint for both the coarse granularity and fine granularity task sets. The effect of granularity is easily seen in Figure 6.1.

The processing time of the tasks, p_j, were estimated based on the number and type of operations performed. Data load and store operations and conditional branching operations were estimated to require 2 cycles while simple data operations were estimated to require 1 cycle. The rejection costs of the tasks, e_j, were chosen to give an indication of the hardware area requirements for the tasks. Thus, just as for the processing times, they were estimated based upon the number and type of operations performed. The weights of the tasks, w_j, were chosen to allow the tasks to be five time units late before being rejected was more profitable. Thus, $w_j = e_j/5$.

The use of fine granularity tasks yielded consistently better solutions than did the use of coarse granularity tasks. At each of the data points, my scheduling algorithm determined the optimal solution for the given task set.

6.3 Multiple Processing Elements

In addition to the single-processor single-ASIC target architecture, my scheduling-based approach can be applied to systems where there are multiple Processing Elements (PEs) and a subset of them must be chosen as the target architecture. The straightforward manner of applying my scheduling-based approach is to choose an initial PE and call it the 'processor.' Run the scheduling with rejection problem with this target processor. The results of this problem yield a scheduled set of tasks that

are to be implemented on the chosen PE and a rejected set of tasks. The release times and deadlines of the rejected tasks can be recalculated based on their dependencies with the scheduled tasks. The rejected tasks then form the input to a second iteration of the scheduling with rejection problem. A second PE is chosen to act as the processor, and the scheduling problem is solved with only the rejected tasks as input. This process is repeated until either no feasible schedule can be found, or all tasks have been scheduled on some PE. I used this approach to solve several examples from the literature.

My scheduling algorithm was prototyped in Java. My results were obtained on a 300 MHz Pentium II system with 128 MB of main memory running the Windows NT 4.0 operating system and Java JDK 1.2. I compare my results with the published results of MOGAC [17], Oh and Ha [51], SOS [55], and Yen and Wolf [64]. All CPU times are given in seconds.

MOGAC [17] is a genetic algorithm-based hardware-software system that allocates and schedules a given set of tasks on a given set of PEs. MOGAC consists of approximately 18,000 lines of C++ and Bison code. Its results were obtained on a 200 MHz Pentium Pro system with 96 MB of main memory running the Linux operating system. The generality of the genetic algorithm solution method allows MOGAC to easily formulate all hardware-software partitioning problem characteristics and constraints. This generality also prevents MOGAC from performing intelligent pruning of the search space to decrease the running time of the algorithm.

Oh and Ha [51] use a heterogeneous multiprocessor list scheduling-based algorithm to perform allocation and scheduling of a given set of tasks on a given set of PEs. Their algorithm is implemented in C++, and their results were obtained on an Ultrasparc I with a 200 MHz processor and 256 MB of main memory. Oh and Ha's approach consists of repeatedly choosing a subset of PEs as the target architecture (starting with the lowest cost PE and heuristically adding one additional PE at a time) and then list scheduling the tasks on this set of PEs. This approach is only useful when the set of PEs is well-defined. Furthermore, their algorithm does not model all characteristics and constraints of the partitioning problem. In particular, their algorithm does not

consider timing constraints on the tasks. It schedules the tasks so as to minimize the makespan of the schedule. Timing constraints are checked only after the schedule has been generated for a set of PEs.

SOS [55] is a Mixed Integer Linear Programming (MILP) formulation for the allocation and scheduling subproblems of the hardware-software partitioning problem. SOS's results were obtained on a Solbourne Series5e/900 (similar to a SPARC 4/490) with 128 MB of main memory. The MILP formulation finds an optimal solution to the combined allocation and scheduling problem; however, its time complexity makes it impractical for large problems.

Yen and Wolf [64] use a gradient-search algorithm to perform allocation and scheduling of a given set of tasks on a given set of PEs. Their algorithm is implemented in C++, and their results were obtained on a Sun Sparcstation SS20. The iterative improvement nature of Yen's algorithm allows it to find locally optimal solutions, but does not guarantee that it will find globally optimal solutions. Therefore, the algorithm's final solution is strongly dependent upon the quality of the initial solution. Thus, the algorithm is useful mainly as a final optimization pass given the solution found by some other approach.

Table 6.2 compares the performance of my approach to that of SOS, MOGAC, and Oh and Ha when they are applied to Prakash and Parker's task graphs [55]. The performance number shown by each task graph is the worst-case finish time, or makespan, of the task graph. For example, "Prakash & Parker 1 (4)," refers to Prakash and Parker's first task graph with a makespan of 4 time units. The cost of

Example (makespan)	No. of Tasks	SOS		MOGAC		Oh & Ha		My Scheduling Approach	
		Cost	CPU time(s)	Cost	CPU time(s)	Cost	CPU time(s)	Cost	CPU time(s)
Prakash & Parker 1 (4)	4 (16)	7	28	7	3.3	7	0.01	7	0.026
Prakash & Parker 1 (7)	4 (16)	5	37	5	2.1	5	0.01	5	0.026
Prakash & Parker 2 (8)	9 (9)	7	4,511	7	2.1	7	0.01	7	0.014
Prakash & Parker 2 (15)	9 (9)	5	385,012	5	2.3	5	0.01	5	0.014

Table 6.2: Prakash and Parker's examples.

a solution is determined by the price of the PEs used in the solution, plus 1 for each communication link required.

In these graphs, an unconventional model for communication is used. A task may begin executing before all of its input data have arrived, and it may output data before it has completed execution. I converted their specifications into graphs which conform to the conventional communication model, i.e., a task can only begin execution when all of its input data have arrived and can only output its data when it is has completed execution. Their model implies that part of each task is independent of the task's input data and may perform some operations that do not yield any output data. This is expressed by splitting each task into multiple tasks such that the resulting tasks conform to the standard notion of communication. The resulting number of tasks is shown in parenthesis next to the original number of tasks in the problem.

All of the approaches were able to find the optimal solutions for these simple examples. The differences between these approaches is evident in their respective running times. The exact MILP approach of SOS causes it to have an exorbitant running time for these small examples. The general random search technique employed by MOGAC is able to find the optimal solution in only a few seconds, and both the approach of Oh and Ha and my approach take only a few milliseconds to find the optimal solution.

Table 6.3 compares the performance of my scheduling-based approach to that of Yen's system, MOGAC, and Oh and Ha's approach when each is run on the clustered and unclustered versions of Hou's task graphs [30]. Task clustering is the process

Example (makespan)	No. of Tasks	Yen		MOGAC		Oh & Ha		My Scheduling Approach	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
Hou 1 & 2 (u)	20	170	10,205.3	170	5.7	170	0.1	170	0.083
Hou 1 & 3 (u)	20	240	11,550.2	N.A.	N.A.	170	0.1	170	0.084
Hou 3 & 4 (u)	20	210	7,135.0	170	8.0	170	0.1	170	0.083
Hou 1 & 2 (c)	8	170	14.96	170	5.1	170	0.01	170	0.045
Hou 1 & 3 (c)	7	170	4.92	N.A.	N.A.	N.A.	N.A.	170	0.042
Hou 3 & 4 (c)	6	170	3.34	170	2.2	N.A.	N.A.	170	0.035

Table 6.3: Hou's examples.

of using a pre-pass to collapse multiple tasks into a cluster of tasks. This cluster is treated as a single task during allocation, i.e., all tasks in a cluster are executed on the same PE. Hou ran Yen's system on the clustered and unclustered versions of his graphs. I use the same clusters as Hou, MOGAC, and Oh and Ha when comparing my results with theirs.

All approaches, except for Yen's approach, found the optimal solution for all of the examined task graphs. As with the small examples of Prakash and Parker, MOGAC required a few seconds while the approach of Oh and Ha and my approach required a few tens of milliseconds. Yen's approach proved to have poor scalability with these examples. The small clustered examples have on the order of ten tasks, and Yen's algorithm was able to find the optimal solutions in a few seconds. The unclustered examples have on the order of twenty tasks, and Yen's algorithm was not able to find the optimal solution for all examples in several thousand seconds of execution time.

Table 6.4 compares the performance of my scheduling-based approach to that of Yen's system, MOGAC, and Oh and Ha's approach when each is applied to Yen's large random task graphs [64]. Yen's random 1 has six independent task graphs, each containing approximately eight tasks. There are eight PE types available. Yen's random 2 consists of eight independent task graphs, each containing approximately eight tasks. There are twelve PE types available. Both examples have zero communication delay and zero communication link cost.

My approach and Oh and Ha's approach were able to find the optimal solution for both of these examples. MOGAC was able to find the optimal solution for Yen's random 2 but failed to find the optimal solution for Yen's random 1. The running time for MOGAC was on the order of a few seconds showing only a slight increase

Example	No. of Tasks	Yen		MOGAC		Oh & Ha		My Scheduling Approach	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
Yen Random 1	50	281	10,252	75	6.4	51	2.1	51	0.699
Yen Random 2	60	637	21,979	81	7.8	81	3.6	81	0.826

Table 6.4: Yen's large random examples.

Example	No. of Tasks	MOGAC		Oh & Ha		My Scheduling Approach	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
MOGAC Random 1	510	39	2,454	39	17.6	39	1.302
MOGAC Random 2	990	35	12,210	13	299.8	13	2.486

Table 6.5: MOGAC's very large random examples.

in running time over the slightly smaller examples of Hou. Oh and Ha's approach required a few seconds to find the optimal solutions, while my approach required only a few tenths of a second.

Table 6.5 compares the performance of my scheduling-based approach to that of MOGAC and Oh and Ha's approach when each is applied to MOGAC's very large random task graphs [17]. MOGAC's random 1 contains eight independent task graphs, each containing approximately sixty-three tasks. There are eight PE types and five link types. MOGAC's random 2 contains ten independent task graphs, each containing approximately ninety-nine tasks. There are twenty PE types and ten link types.

My approach and Oh and Ha's approach were able to find the optimal solution for both of these examples. MOGAC was able to find the optimal solution for MOGAC random 1 but failed to find the optimal solution for MOGAC random 2. The running time for MOGAC was on the order of a few thousand seconds. This running time is over three orders of magnitude more than that required to solve Yen's examples that are only one order of magnitude smaller in terms of number of tasks. Oh and Ha's approach also exhibited an unusually large increase in running time for these large examples. To solve MOGAC random 1, Oh and Ha's approach required over seventeen seconds, and for MOGAC random 2, their approach required nearly three hundred seconds. My approach required only 1.302 seconds to solve MOGAC random 1 (with 510 tasks) and 2.486 seconds to solve MOGAC random 2 (with 990 tasks).

Given the success of my scheduling-based approach in finding the optimal solution to all of these problem instances, it is natural to ask how this was achieved. After all, the problem being solved is strongly $\mathcal{NP}$-hard, and I do not use an exact approach.

Some insight is obtained by examining the solutions to these problems. For all of the problem instances, the optimal solution required either one or two processing elements. Furthermore, none of the examples has hard to satisfy timing constraints. These two features of the optimal solutions allow my simple-minded approach to find the optimal solutions. My approach should perform less well on problem instances containing tight timing constraints and requiring more than two processing elements in an optimal solution.

Part II

Complexity of the Scheduling Problem Formulation

There is no such word as 'impossible' in my dictionary. In fact, everything between 'herring' and 'marmalade' appears to be missing.

Douglas Adams, *Dirk Gently's Holistic Detective Agency*

Chapter 7

The Complexity of Scheduling with Rejection

"Now, here is a game that they like,"
Said the cat.
"They like to fly kites,"
Said the Cat in the Hat.
⋮
Thing Two and Thing One!
They ran up! They ran down!
On the string of one kite
We saw Mother's new gown!

Dr. Seuss, *The Cat in the Hat*

This chapter examines the complexity of the scheduling with rejection problem, probing the boundary between easily solved problems involving rejection and their more difficult generalizations. Knowing that a problem is $\mathcal{NP}$-hard encourages the designer to use heuristics to find a feasible solution to the problem. In addition, if a special case of the problem is solvable in polynomial time, then the design may be specified so as to take advantage of the efficient algorithms that can solve those special cases.

7.1 Introduction

In this chapter, we consider the complexity of the single machine scheduling with rejection problem for various objective functions. Figure 7.1 illustrates the complexity hierarchy describing the complexity relationship between scheduling problems that differ only in their objective function. This hierarchy indicates that a problem with the objective function at the head of an arc reduces to a problem with the objective function at the tail of the arc, e.g., $\alpha\ |\beta\ |C_{\max}$ reduces to $\alpha\ |\beta\ |L_{\max}$. Thus, if $\alpha\ |\beta\ |L_{\max}$ is solvable in polynomial time, then $\alpha\ |\beta\ |C_{\max}$ is solvable in polynomial time. Conversely, if $\alpha\ |\beta\ |C_{\max}$ is $\mathcal{NP}$-hard, then $\alpha\ |\beta\ |L_{\max}$ is $\mathcal{NP}$-hard.

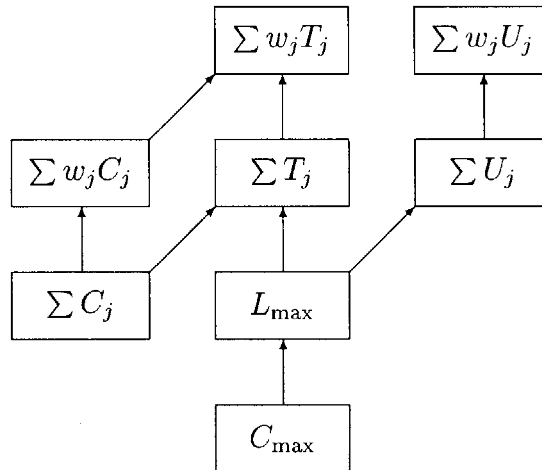


Figure 7.1: Complexity hierarchy for scheduling problems that differ only in their objective functions. An arrow from γ_1 to γ_2 indicates that $\alpha\ |\beta\ |\gamma_1 \ \propto\ \alpha\ |\beta\ |\gamma_2$.

The results presented in this chapter are summarized in Table 7.1. These results narrow the boundary between known polynomial time solvable problems and known $\mathcal{NP}$-hard problems. Figure 7.2 graphically illustrates the boundary with regard to possible objective functions when rejection is allowed. Figure 7.3 graphically illustrates the boundary with regard to possible objective functions when all tasks have a common deadline. Each circle in these figures represents a scheduling problem. A filled circle represents a known $\mathcal{NP}$-hard problem. An empty circle represents

Problem	*Result*	*Presented*
$Pm\|\|(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$	$\mathcal{NP}$-hard proof	Section 7.2.1
$1\|\|(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(nE)$ algorithm	Section 7.2.2
$1\|\|(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(nW)$ algorithm	Section 7.2.2
$1\|\|(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(nP)$ algorithm	Section 7.2.2
$1\|d_j = d\| \sum c_jU_j$	$\mathcal{NP}$-hard proof	Section 7.3.1
$1\|d_j = d\|(\sum_S c_jU_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(nP)$ algorithm	Section 7.3.2
$1\|d_j = d; c_j$ agreeable$\|(\sum_S c_jU_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(n \ln n)$ algorithm	Section 7.3.3
$1\|\|(\sum_S T_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(n^5PE)$ algorithm	Section 7.4.1
$1\|d_j = d; w_j$ agreeable$\| \sum w_jT_j$	$\mathcal{O}(n \ln n)$ algorithm	Section 7.4.2
$1\|d_j = d; w_j$ disagreeable$\| \sum w_jT_j$	$\mathcal{O}(n \ln n)$ algorithm	Section 7.4.2
$1\|d_j = d\|(\sum_S w_jT_j + \sum_S c_jU_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(n(P-d))$ algorithm	Section 7.5.1
$1\|d_j = d\|(\sum_S w_jT_j + \sum_S c_jU_j + \sum_{\bar{S}} e_j)$	FPAS	Section 7.5.2
$1\|d_j = d\|(\sum_S w_jT_j + \sum_S c_jU_j + \sum_{\bar{S}} e_j)$	$\mathcal{O}(n^2dE)$ algorithm	Section 7.5.3

Table 7.1: Results presented in this chapter.

a known polynomial time solvable problem, and a partially filled, or dotted, circle represents a problem with unknown complexity. Boundary problems with new complexity results presented in this dissertation are indicated with a square. Problems whose complexity was previously known are indicated with a double circle.

In addition to the simple complexity results of being either $\mathcal{NP}$-hard or polynomial-time solvable, many of the problems considered in this chapter were proven to be weakly $\mathcal{NP}$-hard. A weakly $\mathcal{NP}$-hard problem is solvable in pseudo-polynomial time. As shown in Table 7.1, many of the results presented in this chapter are pseudo-polynomial time algorithms, proving weak $\mathcal{NP}$-hardness results for many problems.

We examine problems with a common deadline, i.e., $d_j = d$ for all jobs j, since a system may be specified such that all functionality must be completed by a common deadline with the intermediate timing behavior being irrelevant. Furthermore, since

$\mathbf{1}||(\gamma + \sum_{\bar{S}} e_j)$

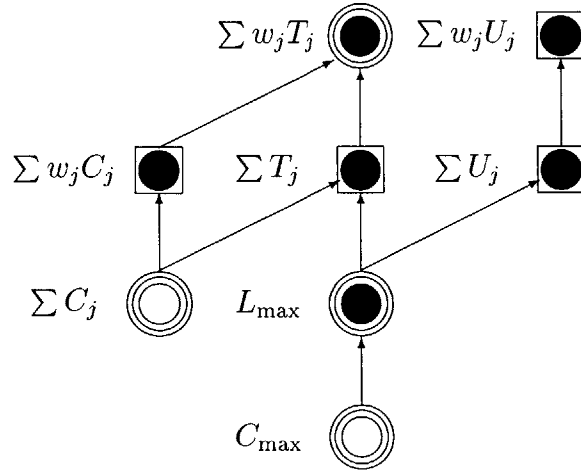


Figure 7.2: Graphical representation of the known complexity boundary for scheduling problems that allow rejection and have no β constraints. Nodes corresponding to problems with new proven complexity bounds presented in this dissertation are boxed. Nodes corresponding to problems whose complexity was previously known are doubly circled. We use the representation given by [24]. Problems are represented by circles, filled-in if known to be $\mathcal{NP}$-hard and empty if known to be in $\mathcal{P}$. An arrow from Π_1 to Π_2 signifies that problem Π_1 is a subproblem of problem Π_2.

$\mathbf{1}|d_j = d|\gamma$

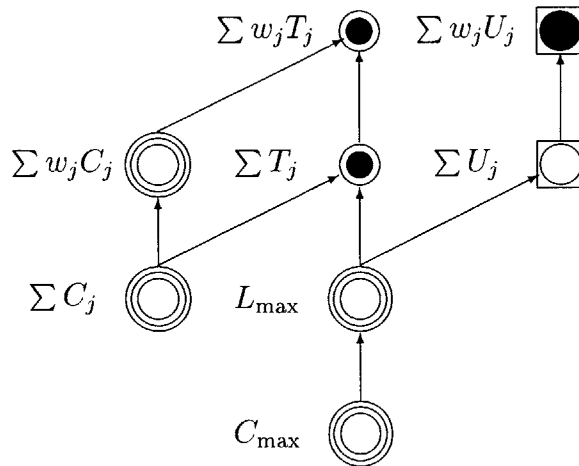


Figure 7.3: Graphical representation of the known complexity boundary for scheduling problems that have a common deadline and have no other β constraints.

my results prove that it is possible to obtain near optimal solutions in polynomial time (with the FPAS presented in Section 7.5.2), it is desirable to specify the system behavior, when possible, with a common deadline.

7.2 The Total Weighted Completion Time with Rejection

In this section, I show that the problem $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is $\mathcal{NP}$-hard in the weak sense. Thus, the problem is $\mathcal{NP}$-hard for any fixed number of machines $m \geq 1$. For any fixed number of machines $m \geq 2$, $Pm||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is $\mathcal{NP}$-hard by restricting the problem to $Pm||\sum w_jC_j$, a known $\mathcal{NP}$-hard problem [11]. On a single machine, the total weighted completion time problem is solvable in polynomial time using Smith's rule [58]. However, I show that allowing rejection makes the single machine problem weakly $\mathcal{NP}$-hard.

7.2.1 Complexity of Total Weighted Completion Time with Rejection

The decision problem formulation of $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is defined as follows.

> Given a set of n independent jobs, $\mathcal{J} = \{J_1, \dots, J_n\}$ with processing times $p_j, \forall 1 \leq j \leq n$, weights $w_j, \forall 1 \leq j \leq n$, and rejection penalties $e_j, \forall 1 \leq j \leq n$, a single machine, and a number K, is there a schedule of a subset of jobs $S \subseteq \mathcal{J}$ on the machine such that $\sum_{j \in S} w_jC_j + \sum_{j \in \bar{S} = \mathcal{J} - S} e_j \leq K$?

I reduce the weakly $\mathcal{NP}$-complete Partition Problem [24] to $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$, proving that even on one machine, scheduling with rejection is $\mathcal{NP}$-complete.

Theorem 7.2.1 *$1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is $\mathcal{NP}$-complete.*

Proof $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is clearly in $\mathcal{NP}$. To prove that it is also $\mathcal{NP}$-hard, I reduce the Partition Problem to it. The Partition Problem is defined as follows:

Given a set $A = \{a_1, a_2, \dots, a_n\}$ of n numbers such that $\sum_{i=1}^{n} a_i = 2b$, is there a subset A' of A such that $\sum_{a_i \in A'} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i = b$?

Given an instance $A = \{a_1, a_2, \dots, a_n\}$ of the Partition Problem, I create an instance of $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ with n jobs, $J_1, \dots, J_n$. Each of the n elements $a_i \in A$ corresponds to a job J_i in $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ with $p_i = w_i = a_i$ and $e_i = ba_i + \frac{1}{2}a_i^2$. Since $p_j = w_j, \forall 1 \leq j \leq n$, the ordering of the scheduled jobs does not affect the value of $\sum_{j \in S} w_jC_j$. Using this fact and substituting for the rejection penalty, the objective function can be rewritten as follows:

$$\begin{aligned}
\sum_{j \in S} w_jC_j + \sum_{j \in \bar{S}} e_j &= \sum_{j \in S} a_j \sum_{(i \leq j, i \in S)} a_i + \sum_{j \in \bar{S}} e_j \\
&= \sum_{j \in S} a_j^2 + \sum_{(i<j, i \in S)} a_i a_j + \sum_{j \in \bar{S}} (ba_j + \tfrac{1}{2}a_j^2) \\
&= \tfrac{1}{2}[(\sum_{j \in S} a_j)^2 + \sum_{j \in S} a_j^2] + b\sum_{j \in \bar{S}} a_j + \tfrac{1}{2}\sum_{j \in \bar{S}} a_j^2 \\
&= \tfrac{1}{2}(\sum_{j \in S} a_j)^2 + b\sum_{j \in \bar{S}} a_j + \tfrac{1}{2}\sum_{j=1}^{n} a_j^2 \\
&= \tfrac{1}{2}(\sum_{j \in S} a_j)^2 + b(2b - \sum_{j \in S} a_j) + \tfrac{1}{2}\sum_{j=1}^{n} a_j^2
\end{aligned}$$

Since $\sum_{j=1}^{n} a_j^2$ is a constant for any particular instance of the Partition Problem, minimizing the objective function to $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is equivalent to minimizing the following function with $x = \sum_{j \in S} a_j$:

$$\frac{1}{2}x^2 + b(2b - x).$$

This function has a unique minimum of $\frac{3}{2}b^2$ at $x = b$, i.e., the optimum solution has $\sum_{j \in S} a_j = b$, if it exists. Therefore, if the optimum solution to $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is equal to $\frac{3}{2}b^2 + \frac{1}{2}\sum_{j=1}^{n} a_j^2$, then there exists a subset, A', of A such that $\sum_{a_i \in A'} a_i = b$, i.e., the answer to the Partition Problem is 'Yes,' and S is a witness. If the optimum solution to $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is greater than $\frac{3}{2}b^2 + \frac{1}{2}\sum_{j=1}^{n} a_j^2$, then there does not exist a partition of A such that $\sum_{a_i \in A'} a_i = b$, i.e., the answer to the Partition Problem is 'No.' Conversely, if the answer to the Partition Problem is 'Yes,' the optimum solution to $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is clearly equal to $\frac{3}{2}b^2 + \frac{1}{2}\sum_{j=1}^{n} a_j^2$.

If the answer to the Partition Problem is 'No,' the optimum solution to $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ is clearly greater than $\frac{3}{2}b^2 + \frac{1}{2}\sum_{j=1}^{n} a_j^2$. ■

7.2.2 Pseudo-Polynomial Time Algorithms

In this section, I give pseudo-polynomial time algorithms for solving $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ exactly. In the scheduling problem $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$, all jobs are assigned either to the set of scheduled jobs S or the set of rejected jobs $\bar{S}$. The simple nature of this scheduling problem allows for straightforward dynamic programs to be developed to optimally solve all instances of the problem. In Section 7.2.2, I give an $\mathcal{O}(n\sum_{j=1}^{n} e_j)$ time dynamic program based on the rejection costs of the rejected jobs. I provide similar dynamic programs based on the weight of the scheduled jobs (Section 7.2.2) and based on the makespan of the scheduled jobs (Section 7.2.2).

Dynamic Programming on the Rejection Costs e_j

I set up a dynamic program that solves the problem $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ when the total rejection cost of the rejected jobs is given. Number the jobs in non-decreasing order of w_j/p_j.

Let $f(j,e)$ denote the optimal value of the objective function when the jobs $1,\dots,j$ are considered and the total rejection cost of the rejected jobs from this set is e. Let $w(j,e)$ denote the total weight of the scheduled jobs that yields the optimal value of $f(j,e)$. Note that:

$$f(j,e) = \begin{cases} 0 & \text{if } j = 0 \text{ and } e = 0, \\ \infty & \text{otherwise.} \end{cases} \tag{7.1}$$

This equation forms the boundary conditions for the dynamic program.

Consider any optimal schedule for the jobs $1,\dots,j$ in which the total rejection cost of the rejected jobs is e. In any such schedule, there are two possible cases: either job j is rejected or job j is scheduled.

Case 1: Job j is rejected. This is possible only if $e \geq e_j$. Otherwise, there is no feasible schedule in which the sum of the rejection costs of the rejected jobs is

e and job j is rejected. Thus, Case 2 must apply. Hence, assume that $e \geq e_j$. The total rejection cost of the rejected jobs among $1, \ldots, j-1$ must be $e - e_j$. Then, the optimal value of the objective function is clearly $f(j-1, e-e_j) + e_j$.

Case 2: Job j is scheduled. Observe that job j is scheduled before all other scheduled jobs from the set $\{1 \ldots j-1\}$, and scheduling it increases all of their completion times by p_j. Therefore, the optimal value of the objective function is $f(j-1, e) + p_j \sum_{i \in S, 1 \leq i \leq j} w_i$. Let $w(j, e) = \sum_{i \in S, 1 \leq i \leq j} w_i$. The use of this variable eliminates the need to recalculate the total weight of the scheduled jobs for every entry $f(j, e)$.

Combining these cases, we have:

$$f(j,e) = \begin{cases} f(j-1,e) + (w(j-1,e) + w_j)p_j & \text{if } e > e_j, \\ \min \left\{ \begin{array}{l} (f(j-1,e) + (w(j-1,e) + w_j)p_j), \\ (f(j-1, e-e_j) + e_j) \end{array} \right\} & \text{otherwise.} \end{cases} \quad (7.2)$$

Observe that the total rejection cost of the rejected jobs can be at most $E = \sum_{j=1}^{n} e_j$. The problem is solved by the calculation of $\min_{0 \leq e \leq E}\{f(n, e)\}$. Thus, we need to compute at most nE values $f(j, e)$. Computation of each value requires $\mathcal{O}(1)$ time due to the use of the $w(j, e)$ values. Therefore, the overall running time of the dynamic program is $\mathcal{O}(nE)$. Moreover, the dynamic program requires $\mathcal{O}(nE)$ space.

Theorem 7.2.2 *The above dynamic programming algorithm exactly solves the problem* $1||(\sum_{j \in S} w_j C_j + \sum_{j \in \bar{S}} e_j)$ *in* $\mathcal{O}(n \sum_{j=1}^{n} e_j)$ *time.*

Dynamic Programming on the Weights w_j

Instead of using the rejection costs to set up the dynamic program, the task weights w_j may be used in a dynamic program. I set up a dynamic program that minimizes the objective function when the total weight of the scheduled jobs is given. Number the jobs in non-decreasing order of p_j/w_j. For any given set of scheduled jobs S, scheduling the jobs in non-decreasing order of p_j/w_j (Smith's rule) minimizes $\sum_{j=1}^{n} w_j C_j$.

Let $f(j,w)$ denote the optimal value of the objective function when the jobs $j,\ldots,n$ are considered and the total weight of the scheduled jobs from this set is w. Note that

$$f(j,w) \quad = \quad \begin{cases} w_n p_n & \text{if } w = w_n \text{ and } j = n, \\ e_n & \text{if } w \neq w_n \text{ and } j = n, \\ \infty & \text{otherwise.} \end{cases} \tag{7.3}$$

This equation forms the boundary conditions for the dynamic program.

Consider any optimal schedule for the jobs $j,\ldots,n$ in which the total weight of the scheduled jobs is w. In any such schedule, there are two possible cases: either job j is rejected or job j is scheduled.

Case 1: Job j is rejected. Then, the optimal value of the objective function is clearly $f(j+1,w)+e_j$, since the total weight of the scheduled jobs among $j+1,\ldots,n$ must be w.

Case 2: Job j is scheduled. This is possible only if $w > w_j$. Otherwise, there is no feasible schedule in which the sum of the weights of the scheduled jobs is w and job j is scheduled. Thus, Case 1 applies. Hence, assume that $w \geq w_j$. Then, the total weight of the scheduled jobs among $j+1,\ldots,n$ must be $w-w_j$. Also, job j is scheduled before all jobs in the optimal schedule among the scheduled jobs in $j+1,\ldots,n$; therefore, the completion time of every scheduled job among $j+1,\ldots,n$ is increased by p_j. Therefore, the optimal value of the objective function is $f(j+1,w-w_j)+wp_j$.

Combining these cases, we have:

$$f(j,w) \quad = \quad \begin{cases} f(j+1,w)+e_j & \text{if } w < w_j, \\ \min\left\{ \begin{array}{l} f(j+1,w)+e_j, \\ f(j+1,w-w_j)+wp_j \end{array} \right\} & \text{otherwise.} \end{cases} \tag{7.4}$$

Observe that the weight of the scheduled jobs can be at most $W = \sum_{j=1}^{n} w_j$. The problem is solved by the calculation of $\min_{0 \leq w \leq W}\{f(1,w)\}$. Thus, we need to compute at most nW values $f(j,w)$. Computation of each value requires $\mathcal{O}(1)$ time.

Therefore, the overall running time of the dynamic program is $\mathcal{O}(nW)$. Moreover, the dynamic program requires $\mathcal{O}(nW)$ space.

Theorem 7.2.3 *The above dynamic programming algorithm exactly solves the problem* $1||(\sum_{j\in S} w_jC_j + \sum_{j\in \bar{S}} e_j)$ *in* $\mathcal{O}(n\sum_{j=1}^{n} w_j)$ *time.*

Dynamic Programming on the Processing Times p_j

In this section, I give another dynamic program that solves $1||(\sum_S w_jC_j + \sum_{\bar{S}} e_j)$ in $\mathcal{O}(n\sum_{j=1}^{n} p_j)$ time. I set up a dynamic program that minimizes the objective function when the total weight of the scheduled jobs, i.e., the makespan, is given. As before, number the jobs in non-decreasing order of p_j/w_j.

Let $f(j,t)$ denote the optimal value of the objective function when the jobs $1,\dots,j$ are considered and the total makespan of the scheduled jobs from this set is t. The boundary conditions for this dynamic program are as follows:

$$f(j,t) = \begin{cases} w_1p_1 & \text{if } t = p_1 \text{ and } j = 1, \\ e_1 & \text{if } t \neq p_1 \text{ and } j = 1, \\ \infty & \text{otherwise.} \end{cases} \tag{7.5}$$

Consider any optimal schedule for the jobs $1,\dots,j$ in which the total makespan of the scheduled jobs is t. In any such schedule, there are two possible cases: either job j is rejected or job j is scheduled.

Case 1: Job j is rejected. Then, the optimal value of the objective function is clearly $f(j-1,t)+e_j$, since the total processing time of the scheduled jobs among $1,\dots,j-1$ must be t.

Case 2: Job j is scheduled. This is possible only if $t \geq p_j$. Otherwise, there is no feasible schedule in which the makespan is t and job j is scheduled. Thus, Case 1 applies. Hence, assume that $t \geq p_j$. The completion time of job j is t, since the jobs in S are scheduled in non-increasing order of w_j/p_j. The total processing time of the scheduled jobs among $1,\dots,j-1$ must be $t-p_j$. Therefore, the optimal value of the objective function is $f(j-1,t-p_j)+w_jt$.

Combining these cases, we have:

$$f(j,t) = \begin{cases} f(j-1,t)+e_j & \text{if } t < p_j \\ \min\left\{ \begin{array}{l} f(j-1,t)+e_j, \\ f(j-1,t-p_j)+w_j t \end{array} \right\} & \text{otherwise.} \end{cases} \qquad (7.6)$$

Observe that the total processing time of the scheduled jobs can be at most $P = \sum_{j=1}^{n} p_j$. The problem is solved by the calculation of $\min_{0 \le t \le P}\{f(n,t)\}$. Thus, we need to compute at most nP values $f(j,t)$. Computation of each value requires $\mathcal{O}(1)$ time. Therefore, the overall running time of the dynamic program is $\mathcal{O}(nP)$. Moreover, the dynamic program requires $\mathcal{O}(nP)$ space.

Theorem 7.2.4 *The above dynamic programming algorithm exactly solves the problem* $1||(\sum_{j\in S} w_j C_j + \sum_{j\in \bar{S}} e_j)$ *in* $\mathcal{O}(n\sum_{j=1}^{n} p_j)$ *time.*

7.3 The Weighted Number of Tardy Jobs with Rejection

When the jobs have deadlines by which they must complete, the total weighted completion time objective is not of much interest. In the presence of deadlines, it is often of interest to schedule the jobs so that as few jobs as possible are tardy, i.e., minimize the number of jobs that complete after their respective deadlines. If the jobs also have a cost associated with being tardy, then it is desirable to minimize the total cost of the tardy jobs. In the traditional scheduling problem notation, these objective functions are denoted as $\sum U_j$ and $\sum c_j U_j$ respectively, where U_j is equal to 1 if job j is tardy and 0 otherwise.

The total (weighted) number of tardy jobs objective function arises when there is no residual value in completing a job soon after its deadline, but the job must be completed. Therefore, all tardy jobs may be sequenced at the end of the schedule in arbitrary order.

7.3.1 Complexity of Total Weighted Number of Tardy Jobs with Rejection

The general weighted scheduling problem $1||\sum c_jU_j$ is weakly $\mathcal{NP}$-hard [39][35]. The special case with all deadlines equal to d is equivalent to the Knapsack problem. This equivalence has been stated previously without proof [54]. I provide a proof of this equivalence by reducing the Knapsack problem to the scheduling problem $1|d_j = d|\sum c_jU_j$.

Theorem 7.3.1 *The decision version of the scheduling problem $1|d_j = d|\sum c_jU_j$ is weakly $\mathcal{NP}$-complete.*

Proof The decision problem is clearly in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete I reduce the known $\mathcal{NP}$-complete Knapsack problem to it. The Knapsack problem is stated as follows:

> Given a finite set A of n items, with each item $j \in A$ having a size s_j and value v_j associated with it, and positive integers B and K. Is there a subset $A' \subseteq A$ such that $\sum_{j\in A'} s_j \leq B$ and such that $\sum_{j\in A'} v_j \geq K$?

The reduction is straightforward. For each element $j \in U$ create a job J_j with $p_j = s_j$ and $c_j = v_j$. Set the deadline $d = B$. The decision question asks: Is there a schedule of jobs $J_1, \dots, J_n$ such that $\sum c_jU_j \leq \sum c_j - K$? The early jobs, i.e., those jobs that complete before the deadline $d = B$, in a valid schedule of the jobs $J_1, \dots, J_n$ correspond to those items chosen to be placed in the knapsack.

To verify that this reduction is valid, consider an optimal schedule of the n jobs. Let S be the set of jobs that complete before the deadline. Clearly, $\sum_S p_j = \sum_S s_j \leq B$ and $\sum_S c_j = \sum_S a_j \geq K$, and the set S corresponds to the knapsack items.

Conversely, let set S be a set of items that corresponds to an optimal solution to the Knapsack problem. Scheduling the corresponding set of jobs

before the deadline d yields an optimal solution to the scheduling problem. Therefore, the decision version of the problem $1|d_j = d|\sum c_jU_j$ is $\mathcal{NP}$-complete.

The pseudo-polynomial time dynamic programming algorithm due to Lawler and Moore [39] for the problem $1||\sum c_jU_j$ optimally solves the problem when all deadlines are equal. Therefore, $1|d_j = d|\sum c_jU_j$ is weakly $\mathcal{NP}$-complete. ■

It follows from this theorem that the total weighted number of tardy jobs with rejection scheduling problem is $\mathcal{NP}$-complete.

Corollary 7.3.1 *The scheduling problem with rejection,* $1|d_j = d|(\sum_{j\in S} c_jU_j + \sum_{j\in \bar{S}} e_j)$*, is* $\mathcal{NP}$*-complete.*

7.3.2 Pseudo-Polynomial-Time Algorithm

In this section, I present a pseudo-polynomial time dynamic program that optimally solves the problem $1||(\sum_{j\in S} c_jU_j + \sum_{j\in \bar{S}} e_j)$ in $\mathcal{O}(n\sum_{j=1}^{n} p_j)$ time. We note that the problem requires us to partition the jobs into two classes: those jobs which are early and those which are either tardy or rejected. We can assume that all early jobs are sequenced in order of their deadlines with the tardy jobs following them in arbitrary order. In this way, the tardy jobs are equivalent to the rejected jobs. Thus, the tardy weight for a job j is $w_j = \min(c_j, e_j)$. If a job is tardy in a schedule using w_j, then it is rejected if $c_j > e_j$.

This observation gives rise to a simple dynamic program for scheduling with rejection. This dynamic program is a modification of the one given by Lawler and Moore [39].

Arrange the jobs in earliest deadline first order. Let $f(j,t)$ be the minimal total cost for the first j jobs subject to the constraint that job j is completed no later than time t. Note that:

$$f(j,t) = \begin{cases} 0 & \text{if } j = 0 \text{ and } t \geq 0, \\ \infty & \text{otherwise.} \end{cases} \tag{7.7}$$

This equation forms the boundary condition for the dynamic program.

Consider an optimal schedule for the job $1, \ldots, j$ in which the completion time of job j, if it is scheduled, is at most t. In any such schedule, there are three possible cases: j is scheduled early and completes before time t, j is scheduled early and completes at time t, and j is rejected or tardy.

Case 1: Job j is scheduled early and completes before time t. Then, the optimal value of the objective function is $f(j, t-1)$, since j has completed at least one time unit before t.

Case 2: Job j is scheduled early and completes at time t. Then, the optimal value of the objective function is $f(j-1, t-p_j)$, since job j completes by its deadline $d_j \geq t$.

Case 3: Job j is rejected or tardy. Then, the optimal value of the objective function is $f(j-1, t) + \min(c_j, e_j)$, since job j does not complete by its deadline it can be rejected (if $e_j \leq c_j$) or scheduled as late as possible after all early jobs have been scheduled.

$$f(j,t) \quad = \quad \min \left\{ \begin{array}{l} f(j, t-1), \\ f(j-1, t-p_j), \\ f(j-1, t) + \min(c_j, e_j) \end{array} \right\} \tag{7.8}$$

Observe that the maximum completion time t that we need to consider is $P = \sum_{j=1}^{n} p_j$. The problem is solved by the calculation of $f(n, P)$. Thus, we need to compute at most nP values $f(j,t)$. Computation of each value requires $\mathcal{O}(1)$ time. The overall computation time required for this dynamic program is $\mathcal{O}(nP)$. Moreover, this algorithm requires $\mathcal{O}(nP)$ space.

Theorem 7.3.2 *The above dynamic programming algorithm exactly solves the problem* $1||(\sum_{j \in S} c_j U_j + \sum_{j \in \bar{S}} e_j)$ *in* $\mathcal{O}(n \sum_{j=1}^{n} p_j)$ *time.*

7.3.3 Special Cases Solvable in Polynomial Time

In this section, we consider the problem where all costs are *agreeable.* The costs are agreeable if $p_i < p_j$ implies $c_i \geq c_j$ and $e_i \geq e_j$. When there is a common deadline d and all costs are agreeable, the problem $1|d_j = d|(\sum_{j \in S} c_j U_j + \sum_{j \in \bar{S}} e_j)$ is solvable in polynomial time by scheduling the jobs in Shortest Processing Time (SPT) first order.

Theorem 7.3.3 *The total weighted number of tardy jobs scheduling problem with common deadline and agreeable weights,* $1|d_j = d; c_j$ *agreeable*$|(\sum_{j \in S} c_j U_j + \sum_{j \in \bar{S}} e_j)$, *is solvable in* $\mathcal{O}(n \ln n)$ *time by scheduling the scheduled jobs in SPT order with ties broken by largest cost first.*

Proof Let the jobs be numbered in SPT order such that $p_j \leq p_{j+1}$, ties broken by smallest cost $w_j = \min(c_j, e_j)$, and let all costs by agreeable. We will consider the cost incurred by a job j not completing before the deadline d to be $w_j = \min(c_j, e_j)$. Thus, a rejected job j is equivalently tardy with cost equal to $e_j \leq c_j$. The resulting problem is equivalent to $1|d_j = d| \sum w_j U_j$.

Let S be an optimal schedule to some instance I of the scheduling problem $1|d_j = d| \sum w_j U_j$. Note that S is an optimal schedule to $1|d_j = d|(\sum_{j \in S} c_j U_j + \sum_{j \in \bar{S}} e_j)$ for the same instance I with $w_j = \min(c_j, e_j)$ and all tardy jobs with $w_j = e_j$ correspond to rejected jobs. Assume that the jobs in S are not in SPT order. The order of the early jobs does not affect the objective function; therefore, assume, without loss of generality, that the early jobs are arranged in SPT order. The order of the tardy jobs does not affect the objective function; therefore, assume, without loss of generality, that the tardy jobs are arranged in SPT order.

Assume that there is at least one tardy job. Let job j be the latest scheduled job in S that has a smaller job $i < j$ scheduled after it. Let i be the earliest scheduled job after job j in S such that $i < j$. By assumption on the ordering of the early jobs, i must be tardy and jobs i and j are

adjacent in the schedule. Interchange jobs i and j to obtain schedule S'. The completion time of job j in S' is equal to the completion time of job i in S. If $p_j = p_i$, then interchanging jobs i and j can only affect the cost of the schedule if job j is early and $w_j \neq w_i$. By the ordering of the jobs, $w_j \leq w_i$; therefore, if $w_j < w_i$, job j must be tardy in the optimal schedule S. Otherwise, interchanging jobs i and j creates a schedule with smaller objective function.

Since $p_j > p_i$, the completion time of job i in S' is less than the completion time of job j in S. Therefore, the number of tardy jobs may only decrease by this swapping. Furthermore, since $w_j \leq w_i$ by the assumption of agreeable costs, the objective function may only decrease by swapping jobs i and j in the schedule.

By repeatedly swapping jobs and then arranging the early and tardy jobs in SPT order, an optimal schedule S' with all jobs in SPT order is obtained. The SPT order may be obtained by sorting the tasks based on processing time. Sorting requires $\mathcal{O}(n \ln n)$ time. Those tardy jobs that have $e_j < c_j$ correspond to rejected jobs. Therefore, the total weighted number of tardy jobs scheduling problem with rejection, common deadline, and agreeable costs, $1|d_j = d|(\sum_{j \in S} c_j U_j + \sum_{j \in \bar{S}} e_j)$, is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in SPT order with ties broken by largest w_j first. ■

7.4 The Total Weighted Tardiness with Rejection

7.4.1 Complexity with Arbitrary Deadlines

The total tardiness scheduling problem $1||\sum T_j$ is weakly $\mathcal{NP}$-hard [41] [18]. Therefore, the same problem with rejection, i.e., $1||(\sum_S T_j + \sum_{\bar{S}} e_j)$, is also $\mathcal{NP}$-hard. To prove that it is weakly $\mathcal{NP}$-hard when the weights are agreeable, I show that an optimal solution to the problem can be found by a dynamic programming algorithm

with worst-case running time of $\mathcal{O}(n^5PE)$ or $\mathcal{O}(n^6 p_{\max} e_{\max})$ where $P = \sum_{j=1}^n p_j$, $E = \sum_{j=1}^n e_j$, $p_{\max} = \max_j(p_j)$, and $e_{\max} = \max_j(e_j)$. The dynamic program solves all total weighted tardiness with rejection problems, i.e., $1||\sum_S w_jTj + \sum_{\bar{S}} e_j$, where the weighting of the jobs is *agreeable* in the sense that $p_i < p_j$ implies $w_i \geq w_j$. The dynamic program is based on the dynamic program of Lawler [41] for the scheduling problem $1||\sum T_j$.

We begin with some useful theorems for the total weighted tardiness problem with agreeable weights. These theorems are valid for the scheduled subset of jobs in the total tardiness with rejection problem. These theorems are due to Lawler [41], and their proofs are repeated here for completeness.

Theorem 7.4.1 ([41]) *Let the jobs have arbitrary weights. Let S be any sequence which is optimal with respect to the given deadlines $d_1, d_2, \ldots, d_n$ and let C_j be the completion time of job j for this sequence. Let d'_j be chosen such that*

$$\min(d_j, C_j) \leq d'_j \leq \max(d_j, C_j).$$

Then any sequence S' which is optimal with respect to the deadlines $d'_1, d'_2, \ldots, d'_n$ is also optimal with respect to $d_1, d_2, \ldots, d_n$ (but not conversely).

Proof Let T denote the total weighted tardiness with respect to $d_1, d_2, \ldots, d_n$ and T' denote the total weighted tardiness with respect to $d'_1, d'_2, \ldots, d'_n$. Let S' be any sequence which is optimal with respect to $d'_1, d'_2, \ldots, d'_n$, and let C'_j be the completion time of job j for this sequence. We have

$$T(S) = T'(S) + \sum_j A_j, \quad (7.9)$$

$$T(S') = T'(S') + \sum_j B_j \quad (7.10)$$

where, if $C_j \leq d_j$,

$$A_j = 0$$

$$B_j = -w_j \max(0, \min(C'_j, d_j) - d'_j),$$

and, if $C_j \geq d_j$,

$$A_j = w_j(d'_j - d_j)$$
$$B_j = w_j \max(0, \min(C'_j, d'_j) - d_j).$$

Clearly, $A_j \geq B_j$ and $\sum_j A_j \geq \sum_j B_j$. Moreover, $T'(S) \geq T'(S')$ because S' is assumed to minimize T'. Therefore, the right hand side of equation (7.9) dominates the right hand side of equation (7.10). It follows that $T(S) \geq T(S')$ and S' is optimal with respect to $d_1, d_2, \ldots, d_n$. ∎

Theorem 7.4.2 ([41]) *Suppose the jobs are agreeably weighted. Then there exists an optimal sequence S in which job i precedes job j if $d_i \leq d_j$ and $p_i < p_j$, and in which all on time jobs are in nondecreasing deadline order.*

Proof Let S be an optimal sequence. Suppose i follows j in S, where $d_i \leq d_j$ and $p_i < p_j$. Then a simple interchange of i and j yields a sequence for which the total weighted tardiness is no greater. If i follows j, where $d_i \leq d_j$ and i and j are both on time, then moving j to the position immediately following i yields a sequence for which the total weighted tardiness is no greater. Repeated applications of there two rules yields an optimal sequence satisfying the conditions of the theorem. ∎

Theorem 7.4.3 ([41]) *Suppose the jobs are agreeably weighted and numbered in nondecreasing due date order, i.e., $d_1 \leq d_2 \leq \cdots \leq d_n$. Let job k be such that $p_k = max_j\{p_j\}$. Then there is some integer a, $0 \leq a \leq n-k$, such that there exists an optimal sequence S in which k is preceded by all jobs j such that $j \leq k+a$ and followed by all jobs j such that $j > k+a$.*

Proof Let C'_k be the latest possible completion time of job k in any sequence which is optimal with respect to due dates $d_1, d_2, \ldots, d_n$. Let S be a sequence which is optimal with respect to the deadlines $d_1, d_2, \ldots, d_{k-1}, d'_k = \max(C'_k, d_k), d_{k+1}, \ldots, d_n$, and which satisfies the conditions of Theorem 7.4.2 with respect to these deadlines. Let C_k be the completion time of job k in

sequence S. By Theorem 7.4.1, S is optimal with respect to the original due dates. Hence, by assumption, $C_k \leq d'_k$. Job k cannot be preceded in S by any job j such that $d_j > d'$, else job j would also be on time, in violation of the conditions of Theorem 7.4.2. And, job k must be preceded by all jobs j such that $d_j \leq d'_k$. Let a be chosen to be the largest integer such that $d_{k+a} \leq d'_k$, and the theorem is proved. ∎

Assume the jobs are agreeably weighted and numbered in nondecreasing deadline order. Suppose we wish to find an optimal schedule of jobs $1, 2, \ldots, n$, possibly rejecting some jobs, with processing of the first scheduled job to begin at time t. Let k be the scheduled job with the largest processing time. It follows from Theorem 7.4.3 that, for some a, $0 \leq a \leq n-k$, if job k is scheduled, there exists an optimal scheduled sequence S of the form of:

1. scheduled jobs from the set $\{1, 2, \ldots, k-1, k+1, \ldots, k+a\}$, in some sequence starting at time t, followed by
2. job k, with completion time $C_k(a) = t + \sum_{j \leq k+a, j \in S} p_j$, followed by
3. scheduled jobs from the set $\{k+a+1, k+a+2, \ldots, n\}$, in some sequence, starting at time $C_k(a)$.

It follows that the overall sequence is optimal only if the sequences of the subsets of jobs in (1) and (3) are optimal for starting times t and $C_k(a)$, respectively. For any given subset J of jobs and starting time t, there is a well-defined sequencing problem. An optimal solution for problem J, t can be found recursively from optimal solutions to problems of the form J', t', where J' is a proper subset of J and $t' \geq t$.

The subsets J which enter into the recursion are of a very restricted type. Each subset consists of jobs in an interval $i, i+1, \ldots, j$, with processing times strictly less than some value p_k. Accordingly, denote such a subset by

$$J(i, j, k) \quad = \quad \{j' | i \leq j' \leq j, p_{j'} < p_k\},$$

and let $T(J(i,j,k),t,e)$ equal the total weighted tardiness for the optimal sequence of jobs in $J(i,j,k)$, starting at time t, and total rejection penalty of e (where rejected jobs are from $J(i,j,k)$.

By the application of Theorem 7.4.3, we have:

$$\begin{aligned} T(J(i,j,k),t,e) \;=\; & \min\{\min_{a,1\le b\le e}\{T(J(i,k+a,k'),t,b)+ \\ & w_k \max\{0, C_{k'}(a)-d_{k'}\}+ \\ & T(J(k'+a+1,j,k'),C_{k'}(a),e-b)\}, \\ & \min_{a,1\le b\le e-e_{k'}}\{T(J(i,k+a,k'),t,b)+ \\ & T(J(k'+a+1,j,k'),C_{k'}(a),e-b-e_{k'})\}\} \end{aligned} \tag{7.11}$$

where k' is such that

$$p_{k'} = \max\{p_{j'} | j' \in J(i,j,k) \text{ and } k' \text{ not rejected}\},$$

and

$$C_{k'}(a) = t + \sum_{\substack{j' \in J(i,k+a,k') \text{ and} \\ j' \text{ not rejected}}} p_{j'},$$

where the summation is taken over all jobs $j' \in J(i,k_a,k')$.

The initial conditions for the equations 7.11 are

$$\begin{aligned} T(\emptyset,t,e) &= 0 \\ T(j,t,e) &= \begin{cases} \infty & e_j \neq e \neq 0 \\ 0 & e_j = e \neq 0 \\ w_j \max(0, t+p_j-d_j) & e = 0. \end{cases} \end{aligned}$$

Establishing an upper bound on the worst-case running time required to compute an optimal schedule for the complete set of n jobs is straightforward. There are no more than $\mathcal{O}(n^3)$ subsets $J(i,j,k)$. (There are no more than n values for each of the indices, i, j, k. Moreover, several distinct choices of the indices may specify the same subset of jobs.) There are no more than $P = \sum p_j \le np_{\max}$ possible values of t. There are no more than $E = \sum e_j \le ne_{\max}$ possible values of e. Hence, there are

no more than $\mathcal{O}(n^3 PE)$ or $\mathcal{O}(n^5 p_{\max} e_{\max})$ equations to solve. Each equation requires minimization over at most n alternatives (for a) with $\mathcal{O}(n)$ running time. Therefore, the overall running time is bounded by $\mathcal{O}(n^4 PE)$ or $\mathcal{O}(n^6 p_{\max} e_{\max})$.

7.4.2 Complexity with Common Deadline

Theorem 7.4.4 *In any optimal schedule S for $1|d_j = d|\sum w_j T_j + \sum e_j$, the scheduled jobs J_j that are started at or after the common due date d are scheduled in order of nonincreasing values of w_j/p_j.*

Proof Follows immediately from Smith's ratio rule [58]. ■

Assume $d < \sum_j p_j$; otherwise, we can schedule all jobs before d.

Theorem 7.4.5 *In each optimal schedule S for $1|d_j = d|\sum w_j T_j + \sum e_j$, there is no inserted idle time.*

Proof By contradiction. Let S be an optimal schedule for $1|d_j = d|\sum w_j T_j + \sum e_j$ that has inserted idle period of length t. Let J_j be the first job that is scheduled after the idle period. J_j is ready to execute at the beginning of the idle period since we are not considering problems with release times and inter-job lag times. Therefore, job J_j can be started t time units earlier, thereby moving the idle period after job J_j. By starting job J_j t time units earlier, its tardiness will either decrease or remain unchanged.

By repeating this argument for all jobs scheduled after J_j in S, the idle period may be eliminated from the schedule. By our assumption that $d < \sum_j p_j$, there is at least one job J_i that is tardy in S. By eliminating the idle period, the tardiness of this job is decreased by t. Therefore, the schedule S' created by removing the idle period has a cost that is at least t less than the cost of S, contradicting our assumption that S is an optimal schedule with inserted idle time. ■

For the special case when all deadlines are equal, the total weighted tardiness problem $1|d_j = d|\sum w_jT_j$ with agreeable weights is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in Shortest Processing Time (SPT) first order (ties broken by largest weight, w_j, first). Lawler [41] stated this observation without proof.

Theorem 7.4.6 *The total weighted tardiness problem with common deadline and agreeable weights, $1|d_j = d|\sum w_jT_j$, is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in SPT order with ties broken by largest weight first.*

Proof Let the jobs be numbered in SPT order such that $p_j \geq p_{j+1}$, ties broken by smallest weight w_j, and let the jobs be agreeably weighted. Let S be an optimal schedule to some instance I of the scheduling problem $1|d_j = d|\sum w_jT_j$ with agreeable weights. Assume that the jobs in S are not in SPT order. The order of the early jobs does not affect the total tardiness; therefore, assume, without loss of generality, that the early jobs are arranged in SPT order.

By Smith's ratio rule [58], the tardy jobs that start after the deadline d must be in SPT order (with respect to the tardy jobs only).

Assume that there is at least one tardy job. Let job j be the latest scheduled job in S that has a smaller job $i > j$ scheduled after it. Let i be the latest scheduled job after job j in S such that $i > j$. By assumption on the ordering of the early jobs, i must be tardy. Interchange jobs i and j to obtain schedule S'. The completion time of job j in S' is equal to the completion time of job i in S. If $p_j = p_i$, then interchanging jobs i and j will result in a schedule with smaller total weighted tardiness ($w_j < w_i$ due to the ordering tie breaker), contradicting the assumption of optimality of S. Thus, $p_j > p_i$, and the completion time of job i in S' is less than the completion time of job j in S. In addition, all jobs scheduled between jobs i and j in S' have a completion time that is less than their completion time in S. Therefore, by assumption that S is an optimal schedule, i must be the job in S that starts before the deadline d and completes after the

deadline. Furthermore, $w_i = w_j$; otherwise, by interchanging i and j to obtain S', a schedule with smaller total weighted tardiness has been found, contradicting the assumption that S is optimal.

It follows from this argument that the k jobs that start after the deadline d are the k largest jobs, and they are arranged in SPT order. By swapping jobs i and j and then arranging the early jobs in SPT order, an optimal schedule S' with all jobs in SPT order is obtained. The SPT order may be obtained by sorting the tasks based on processing time. Sorting requires $\mathcal{O}(n \ln n)$ time. Therefore, the total weighted tardiness problem with common deadline and agreeable weights, $1|d_j = d|\sum w_j T_j$, is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in SPT order with ties broken by largest weight first. ■

Another special case that is solvable in polynomial time occurs when the weighting of the jobs is *disagreeable* in the sense that $p_j < p_i$ implies $w_j/p_j \leq w_i/p_i$. For the special case when all deadlines are equal, the total weighted tardiness problem $1|d_j = d|\sum w_j T_j$ with disagreeable weights is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in Longest Processing Time (LPT) first order (ties broken by largest weight, w_j, first).

Theorem 7.4.7 *The total weighted tardiness problem with common deadline and disagreeable weights, $1|d_j = d|\sum w_j T_j$, is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in LPT order with ties broken by largest weight first.*

Proof Let the jobs be numbered in LPT order such that $p_j \leq p_{j+1}$, ties broken by smallest weight w_j, and let the weighting of the jobs be disagreeable. Let S be an optimal schedule to some instance I of the scheduling problem $1|d_j = d|\sum w_j T_j$ with disagreeable weights. Assume that the jobs in S are not in LPT order. The order of the early jobs does not affect the total tardiness; therefore, assume, without loss of generality, that the early jobs are arranged in LPT order. By Smith's ratio rule [58], the tardy jobs that

start after the deadline d must be in LPT order (with respect to the tardy jobs only).

Let j be the latest scheduled job in S such that there is a larger job $i > j$ scheduled after it in S. Let i be the first such job in S. By assumption on the ordering of the early jobs, i is tardy. Swap jobs i and j to obtain a new schedule S'. Swapping jobs i and j does not affect the tardiness of any other job in the schedule. Therefore, the difference in cost between schedules S and S', if any, will be due to jobs i and j. Furthermore, there are two possible cases for the choice of jobs i and j: (1) job j is early and job i starts at or before the deadline d in S and (2) job j starts before the deadline d but completes after the deadline and job i starts after the deadline in S.

Case 1: The cost due to jobs j and i in S is $w_i p'_i$, where $0 < p'_i \leq p_i$. The cost due to jobs j and i in S' is $w_j p'_i + w_i \max(0, p'_i - p_j)$. By assumption on the optimality of S

$$w_i p'_i \leq w_j p'_i + w_i \max(0, p'_i - p_j). \tag{7.12}$$

If $p'_i \leq p_j$,

$$w_i p'_i \leq w_j p'_i.$$

Due to the disagreeable weights $w_j \leq w_i$; therefore, $w_i = w_j$; otherwise, S' has a smaller objective function than does S, contradicting the assumption of optimality of S. If $p'_i > p_j$,

$$\begin{aligned} w_i p'_i &\leq w_j p'_i + w_i p'_i - w_i p_j \\ 0 &\leq w_j p'_i - w_i p_j \end{aligned}$$

$$w_i / p'_i \leq w_j / p_j.$$

Due to the disagreeable weights $w_j / p_j \leq w_i / p_i$. Since $p'_i \leq p_i$, $w_j / p_j \leq w_i / p'_i$. Therefore, $w_i / p'_i = w_j / p_j$, and S' is optimal; otherwise, S' has a

smaller objective function than does S, contradicting the assumption of optimality of S.

Case 2: The cost due to jobs j and i in S is $w_i p_i + w_j p'_j$, where $0 < p'_j < p_j$. The cost due to jobs j and i in S' is $w_i(p_i + p'_j - p_j) + w_j p_j$. By assumption on the optimality of S

$$\begin{aligned} w_i p_i + w_j p'_j &\leq w_i(p_i + p'_j - p_j) + w_j p_j \qquad (7.13) \\ w_j p'_j &\leq w_i(p'_j - p_j) + w_j p_j \\ w_i(p_j - p'_j) &\leq w_j(p_j - p'_j). \end{aligned}$$

Due to the disagreeable weights $w_j \leq w_i$; therefore, $w_j = w_i$. Otherwise, S' has a smaller objective function than does S, contradicting the assumption of optimality of S.

After swapping jobs i and j, early jobs and tardy jobs may be rearranged into LPT order. By repeatedly swapping jobs in this manner, an optimal schedule in LPT order may be obtained. Sequencing the jobs into LPT order requires time $\mathcal{O}(n \ln n)$. Therefore, the total weighted tardiness problem with common deadline and disagreeable weights, $1|d_j = d|\sum w_j T_j$, is solvable in $\mathcal{O}(n \ln n)$ time by scheduling the jobs in LPT order with ties broken by largest weight first. ■

Arkin and Roundy [1] proved that for disagreeable job weights and arbitrary deadlines, the problem $1||\sum w_j T_j$, is weakly $\mathcal{NP}$-complete.

7.5 The Total Weighted Tardiness plus Weighted Number of Tardy Jobs with Rejection and Common Deadline

In this section, we consider a variant of the common deadline scheduling problems we have examined thus far. Specifically, we consider the problem $1|d_j = d|(\sum_S w_j T_j +$

$\sum_S c_jU_j + \sum_{\bar{S}} e_j)$, which we will denote as TWTDUE. In this scheduling problem, if a job is scheduled, it has a fixed cost c_j for being tardy, and it incurs a weighted tardiness penalty as well. This problem is $\mathcal{NP}$-hard. In Section 7.5.1, I present a pseudo-polynomial time algorithm for this problem, proving that it is weakly $\mathcal{NP}$-hard. In Section 7.5.2, I modify the pseudo-polynomial time algorithm to obtain a Fully Polynomial Time Approximation Scheme (FPAS). Finally, in Section 7.5.3, I present a second pseudo-polynomial time algorithm for the scheduling problem.

7.5.1 A Simple Pseudo-Polynomial Time Algorithm

Assume the jobs are numbered in nondecreasing order of w_j/p_j. One way to solve the TWTDUE problem is to generate a list of all feasible combinations of TWTDUE and processing times. Each such combination is represented by a pair (Z, P) for which there is a subset of jobs S with

$$\begin{aligned} Z &= \sum_{j \in S'} w_jT_j + \sum_{j \in S'} c_jU_j + \sum_{j \in \bar{S}'} e_j \\ P &= \sum_{j \in S} p_j \leq B = \sum_j p_j - d \end{aligned}$$

where the set S' is the set of scheduled jobs in S, the set $\bar{S}'$ is the set of rejected jobs in S, and the tardiness of a job is determined by scheduling the scheduled jobs in nonincreasing order of w_j/p_j starting at time $\sum_j p_j - \sum_{j \in S} p_j$.

The list can be generated in n iterations as follows. Initially place only the pair $(0, 0)$ in the list. Then, at iteration j form from each pair (Z, P) in the list two 'candidate' pairs, $(Z + w_jT_j + c_jU_j, P + p_j)$ and $(Z + e_j, P + p_j)$, if $P + p_j \leq B$, and place the candidate pair with the smallest Z value in the list if it does not duplicate an existing pair. The result is that at the end of iteration j, each pair in the list represents a feasible tardiness with rejection-processing time combination for some subset of items $S \subseteq \{1, 2, \ldots, j\}$, and each subset is represented by a pair.

This procedure is inefficient because it generates many pairs which are not needed to determine an optimal solution with cost Z^*. It does not affect the computed value of Z^* if 'dominated' pairs are discarded. If (Z, P) is in the list, one may eliminate

any pair (Z', P') where $Z \leq Z'$ and $P \geq P'$. After dominated pairs are eliminated, each remaining pair (Z, P) satisfies the following conditions at the end of iteration j:

- Z is the smallest attainable TWTDUE for a subset of items $S \subseteq \{1, 2, \ldots, j\}$ with total processing time at least P, and
- P is the largest attainable total processing time for a subset of items with TWTDUE at most equal to Z.

The procedure is revised as follows. At the end of each iteration, the pairs (Z, P) are in strictly increasing order of P and of Z. To perform iteration j, produce the candidate pair $(Z', P + p_j)$, with $Z' = \min(Z + w_j T_j + c_j U_j, Z + e_j)$, for each pair (Z, P) provided $P + p_j \leq B$. Since the list is in increasing order of P, the production of candidate pairs can be terminated whenever a pair (Z, P) is reached for which $P + p_j > B$. Then, merge the existing list and the list of candidates discarding dominated pairs in the process. This is easily accomplished because of the ordering of Z's and of P's.

At the end of iteration n there exists a set of (Z, P) pairs such that the addition of some job $i \in \bar{S}$ to S will cause $P + p_j > B$. This is the set of (Z, P) pairs we are interested in since one of these pairs will yield the optimal solution to the TWTDUE problem. To find an optimal solution, start at the end of the list of pairs and for each (Z, P) produce the candidate pair $(Z + w_j T_j + c_j U_j, P + p_j)$ such that job j is scheduled beginning at time $\sum_i p_i - (P + p_j)$ and $w_j(\sum_i p_i - P) + c_j = \min_{k \in X}\{w_k(\sum_i p_i - P) + c_k\}$ where $X = \{k | k \in \bar{S} \text{ and } P + p_k > B\}$. If all of the jobs in S are rejected, then produce the candidate pair $(Z + e_j, P + p_j)$ such that $e_j = \min_{k \in X} e_k$ where $X = \{k | k \in \bar{S} \text{ and } P + p_k > B\}$. Since the list is in increasing order of P and we are traversing the list in reverse order, the production of candidate pairs can be terminated whenever a pair (Z, P) is reached for which $B - P > \max_j p_j$. A candidate pair with the minimum Z' value, where $Z' = Z + w_j T_j + c_j U_j$ if some jobs in S are scheduled and $Z' = \min\{Z + w_j T_j + c_j U_j, Z + e_j\}$ if all jobs in S are rejected, corresponds to an optimal solution to the TWTDUE problem. The optimal schedule for the TWTDUE problem is constructed from $(Z', P + p_j)$ by scheduling

all jobs in $\bar{S} - \{j\}$ in arbitrary order starting at time 0. If job j is not rejected, schedule job j beginning at time $\sum_i p_i - P - p_j$, and schedule the scheduled jobs in S in nonincreasing order of w_j/p_j beginning at time $\sum_i p_i - P$.

At each iteration, the running time and space requirements are linearly proportional to the number of pairs existing in the list at the beginning of the iteration. An upper bound on the number of pairs in the list is $\min(Z^*, B)$, where Z^* is the total cost in an optimal solution. Note that B is an upper bound on the number of pairs in the list. Each of the first n iterations requires time $\mathcal{O}(1)$ per pair (Z, P). The final iteration to find an optimal solution requires $\mathcal{O}(n)$ time per pair (Z, P). Hence, an upper bound on the running time for the overall procedure, in the worst case, is $\mathcal{O}(nB)$, and an upper bound on the space requirements is $\mathcal{O}(n + B)$. (The term n in the space bound accounts for the storage of input data.)

Up to this point, we have ignored the problem of constructing an optimal solution. A straightforward method, employed in [32], is to convert each pair (Z, P) to a triple (Z, P, S), where S is a list of indices of items such that

$$\sum_{j \in S} p_j = P, \sum_{j \in S'} w_j T_j + \sum_{j \in \bar{S}'} = Z.$$

Initially, only the triple $(0, 0, \emptyset)$ is placed in the list. Thereafter, at iteration j, each candidate triple is of the form $(Z + \min(w_j T_j, e_j), P + p_j, S \cup \{j\})$. This has the effect of increasing the space bound to $\mathcal{O}(nB)$, since S may be $\mathcal{O}(n)$ in size. Forming the set $S \cup \{j\}$ may be assumed to require $\mathcal{O}(n)$ time, thereby also adding an extra factor of n to the time bound, making it $\mathcal{O}(n^2 B)$.

A more efficient method, employed in [40], constructs S by 'backtracking' through a secondary data structure in the form of a rooted tree.

Each entry in the list of (Z, P) pairs has five components: a Z value, a P value, a pointer to the next entry in the list, a pointer to the previous entry in the list, and a pointer to a node in the rooted tree. Each tree node has two components: an item index j and a pointer to its parent node.

To find the set S for an entry (Z, P) in the list, one goes to the tree node indicated by its pointer and then reads off the item indices associated with nodes on the path

to the root. It is easily seen that S can be constructed in $\mathcal{O}(n)$ time.

Initially, the list contains only the pair $(0,0)$, and the tree pointer for this pair is directed to the root of the tree. Thereafter, whenever a pair $(Z+\min(w_jT_j, e_j), P+p_j)$ is added to the list of (Z,P) pairs, the tree pointer for $(Z+\min(w_jT_j,e_j), P+p_j)$ is directed to a new tree node with associated item index j. The pointer for this new node is directed to the node pointed to by (Z,P). These operations can be implemented in constant time for each new pair $(Z+\min(w_jT_j,e_j), P+p_j)$, or in $\mathcal{O}(nB)$ time overall. Moreover, the tree requires $\mathcal{O}(nB)$ space.

7.5.2 A Fully Polynomial Time Approximation Scheme

The computation can be made more efficient by reducing the number of distinct P values which may occur in the pairs (Z,P). The simplest method to accomplish this is to replace each c_j coefficient by

$$b_j = \left\lfloor \frac{c_j}{k} \right\rfloor,$$

each e_j coefficient by

$$f_j = \left\lfloor \frac{e_j}{k} \right\rfloor,$$

each p_j coefficient by

$$q_j = \left\lfloor \frac{p_j}{k} \right\rfloor,$$

and replace the deadline d by

$$d' = \left\lfloor \frac{d}{k} \right\rfloor,$$

where k is a suitably chosen scale factor. Then $\frac{B}{k}$ replaces B in the time and space bounds given above.

We note that by using q_j, the tardiness values are affected. Let C' and T' represent the new completion time and tardiness values respectively using q_j instead of p_j. Then,

$$\begin{aligned} \sum_j q_j &= \sum_j \lfloor p_j/k \rfloor \\ &\leq 1/k \sum_j p_j. \end{aligned}$$

Therefore, all of the completion times are scaled by at most $1/k$, i.e., $C'_j \leq \frac{C_j}{k}$, and, consequently,

$$\begin{aligned} T'_j &= C'_j - d' \\ &\leq \frac{C_j}{k} - \frac{d}{k} \\ &\leq \frac{1}{k}(C_j - d) \\ &\leq \frac{T_j}{k}. \end{aligned}$$

requiring that the objective function be scaled by at most $1/k$.

We note the inequality

$$kq_j \leq p_j \leq k(q_j + 1).$$

It follows that for any set S

$$\sum_{j \in S} p_j - k \sum_{j \in S} q_j < k|S|.$$

Hence, if we can insure that

$$k|S^*| \leq \epsilon B,$$

where S^* is an optimal solution, then k will be a valid scale factor: the solution found by the computation outlined in the previous section will be within the prescribed accuracy $\epsilon > 0$.

We know that $|S^*| \leq n$, and, without loss of generality, we can assume that $B \geq p_{\max}$, where $p_{\max} = \max_j(p_j)$. Therefore, we may choose

$$k = \frac{1}{n}\epsilon p_{\max}.$$

Now, $B \leq np_{\max}$, so

$$\frac{B}{k} \leq \frac{n^2}{\epsilon}.$$

Substituting $\frac{n^2}{\epsilon}$ for B in the bounds obtained in the previous section, we obtain time and space bounds of $\mathcal{O}(\frac{n^3}{\epsilon})$.

To prove that the performance ratio, $R(I)$, is less than or equal to $1 + \epsilon$ we first observe

$$\text{OPT}(I) - k\ \text{OPT}(I_k) \leq kn,$$

where $\text{OPT}(I)$ is the optimal solution on instance I and $\text{OPT}(I_k)$ is the optimal solution on instance I with all processing times scaled by $\frac{1}{k}$. This implies that

$$\text{OPT}(I) - A_k(I) \leq kn,$$

where $A_k(I)$ is the solution obtained by the scaling algorithm described above. Note also that $\text{OPT}(I) \geq 2$ by our assumption that $B \geq p_{\max}$. With $k = \frac{1}{n}\epsilon p_{\max}$, we then derive the performance ratio as follows,

$$\begin{aligned} R(I) &= \frac{A_k(I)}{\text{OPT}(I)} \\ &\leq \frac{A_k(I) + kn}{A_k(I)} \\ &= 1 + \frac{kn}{A_k(I)} \\ &\leq 1 + \frac{kn}{\text{OPT}(I) - kn} \\ &\leq 1 + \frac{kn}{2 - kn} \\ &\leq 1 + \epsilon. \end{aligned}$$

Before we state the following theorem, we formally define a fully polynomial approximation scheme.

Definition 7.5.1 *An* approximation scheme *for an optimization problem* Π *is an algorithm* A *which takes as input both the instance* I *and an error bound* ϵ*, and has the performance guarantee*

$$R_A(I, \epsilon) \leq (1 + \epsilon).$$

Definition 7.5.2 *A* fully polynomial approximation scheme *(FPAS) is an approximation scheme* A_ϵ *where each algorithm* A_ϵ *runs in time polynomial in the length of the input instance* I *and* $1/\epsilon$*.*

Theorem 7.5.1 *Algorithm* A_k *described above with* $k = \frac{1}{n}\epsilon p_{\max}$ *is a fully polynomial approximation scheme with error bound* ϵ *for* $1|d_j = d|\sum w_j T_j + \sum e_j$*.*

Proof Follows from the running time bound of $\mathcal{O}(\frac{n^3}{\epsilon})$, the performance ratio $R(I) \leq 1+\epsilon$, and the definition of a fully polynomial approximation scheme.

■

7.5.3 Dynamic Programming on the Rejection Costs

In this section, we will investigate another dynamic program to solve the TWTDUE scheduling problem, $1|d_j = d|(\sum_S w_j T_j + \sum_S c_j U_j + \sum_{\bar{S}} e_j)$. When rejection is allowed, the makespan of the scheduled jobs is not known in advance. There are an exponential number of sets of jobs that may be rejected. With arbitrary processing times, there are an exponential number of possible makespans that will need to be examined.

Notice that when rejection is allowed there are four different classes of jobs: (1) those jobs that finish before the deadline d, (2) those jobs that start at or after the deadline d, (3) those jobs that are rejected, and (4) the job the starts before the deadline d and finishes at or after the deadline (referred to as the *crossing job*).

There is no way to know, a priori, the crossing job. The dynamic program for the scheduling problem without rejection does not need to know what the crossing job is a priori because it has a fixed makespan. With a variable makespan due to the possible rejection of jobs, knowing the crossing job can simplify the problem. Therefore, for each possible crossing job, a dynamic program is solved assuming that the crossing job is scheduled, and the best result is chosen from the possible solutions. The procedure is as follows:

1. Order the jobs in non-decreasing order by w_j/p_j ratio.

2. Solve the following dynamic program for each subset of $n-1$ jobs:

   Let $f(j,t,e)$ be the minimal total cost for the first j jobs subject to the constraint that the latest scheduled job in the set $\{1,\dots,j\}$ completes no later than time t and the total cost of the rejected jobs equals e.

$$\begin{aligned} f(0,t,e) &= 0 && t \geq 0; e \geq 0 \\ f(j,t,e) &= \infty && t < 0; j = 0,\dots,n \\ f(j,t,e) &= \infty && e < 0; j = 0,\dots,n \end{aligned}$$

$$f(j,t,e) = \min \left\{ \begin{array}{l} f(j,t-1,e) \\ f(j-1,t-p_j,e) \\ f(j-1,t,e)+w_jt+c_j \\ f(j-1,t,e-e_j)+e_j \end{array} \right\} \tag{7.14}$$

3. Let $f^{(k)}(n,t,e)$ denote the optimal solution for the subset of jobs $\{1,\ldots,k-1,k+1,\ldots,n\}$. Find the values of k, t, and e which minimize

$$f^{(k)}(n,t,e)+w_k(t+p_k-d), \qquad (d-p_k \le t < d).$$

The values of k and t identify the crossing job and its time of completion, while $f^{(k)}(n,t,e)$ can be used to determine the early jobs, the tardy jobs, and the rejected jobs. Solving the dynamic program requires $\mathcal{O}(ndE)$ time, where $E = \sum_j e_j$ is the sum of the rejection costs for all tasks. The overall computation requires $\mathcal{O}(n^2dE)$ time.

To justify the expression for the tardy jobs, consider the partition of the first j jobs into early jobs and tardy jobs. Let $A_j = \sum_{i=1}^{j} p_i$. If the total processing time of the early jobs is t, the total processing time of the tardy jobs is $A_j - t$. It follows that if job j is tardy, its completion time will be $A_n - (A_j - t)$, and the cost will be $w_j(A_n - (A_j - t)) = w_jt +$ constant. If some of the jobs are rejected, the cost is still $w_jt +$ constant since the ordering of the tardy jobs (excluding the crossing job) is fixed by their ratios w_j/p_j.

7.5.4 Using the FPAS

The FPAS for the TWTDUE problem may be used as the fundamental step in an algorithm for solving the general problem $1||(\sum_S w_jT_j + \sum_{\bar{S}} e_j)$. The algorithm is shown in Figure 7.4. This algorithm incrementally builds a solution to the problem $1||(\sum_S w_jT_j + \sum_{\bar{S}} e_j)$ starting with the last scheduled (and rejected) jobs. At each iteration of the algorithm, all jobs are assumed to have a deadline equal to $d_{\max}$, the maximum deadline of the unscheduled and not rejected jobs. Jobs with a deadline

$d_j < d_{\max}$ are assumed to incur no cost if they complete before $d_{\max}$. However, if job j completes after $d_{\max}$, it incurs a cost that is equal to the cost incurred with a deadline of d_j. This cost is obtained by computing $c_j = w_j(d_{\max} - d_j)$ for each unscheduled and not rejected job j and adding the term c_jU_j to the objective function. The result of these modifications is that, at each iteration, a new problem $1|d_j = d_{\max}|(\sum_S w_jT_j + \sum_S c_jU_j + \sum_{\bar{S}} e_j)$ must be solved. A solution to this common deadline scheduling problem is obtained by the FPAS described Section 7.5.2.

The solution to the FPAS yields a partial solution to the original problem $1||(\sum_S w_jT_j + \sum_{\bar{S}} e_j)$. The rejected tasks in the solution to the FPAS are rejected in the solution to the original problem. However, only the last few scheduled tasks in the solution to the FPAS may form a partial solution to the original problem. All tasks in the solution to the FPAS that complete after $d_{\max}$ are prepended to the partial schedule obtained in previous iterations. Subsequent iterations of the algorithm do not consider the scheduled and rejected tasks that form a partial solution to the original problem. A dummy job 0 with deadline $d_0 = 0$, processing time $p_0 = 0$, and rejection cost $e_0 = \infty$ is added to the task set to ensure that all jobs are either scheduled or rejected by the algorithm.

After all jobs have been either scheduled or rejected, the scheduled jobs are compressed to remove any idle time from the schedule. Compression only decreases the completion times of the scheduled tasks; therefore, compression only decreases the cost of the schedule. The compressed schedule and the set of rejected jobs form a solution to the original problem $1||(\sum_S w_jT_j + \sum_{\bar{S}} e_j)$.

It is an open problem whether or not a Polynomial Time Approximate Scheme (PTAS) exists for the problem $1||(\sum_S w_jT_j + \sum_{\bar{S}} e_j)$.

```
1:   // Incrementally build a solution to 1||(∑_S w_j T_j + ∑_S̄ e_j) starting with the
2:   // last jobs to be scheduled (and rejected).  Set 𝒯 is assumed to have a dummy job
3:   // with deadline 0 and processing time 0.
4:
5:   𝒯 ⟵ set of jobs to be scheduled or rejected
6:   S ⟵ ∅                                        // schedule of scheduled jobs
7:   S̄ ⟵ ∅                                        // set of rejected jobs
8:
9:   while( 𝒯 ≠ ∅ ) {
10:     // Assume all jobs have the maximum deadline d_max of the unscheduled jobs
11:     d_max ⟵ max_{j∈𝒯} d_j
12:     // Compensate for assumption that deadline is now d_max instead of d_j by adding scaling factor
13:     // of c_j U_j to objective function.
14:     c_j ⟵ w_j (d_max − d_j) ∀ j ∈ 𝒯
15:     𝒯' ⟵ 𝒯 such that d_j = d_max
16:     S' ⟵ schedule from FPAS used to solve 1|d_j = d_max|(∑_S w_j T_j + ∑_S c_j U_j + ∑_S̄ e_j)
17:        for input 𝒯'
18:     S̄' ⟵ rejected jobs from FPAS used to solve 1|d_j = d_max|(∑_S w_j T_j + ∑_S c_j U_j +
19:        ∑_S̄ e_j) for input 𝒯'
20:     // Jobs completed after d_max in S' form partial solution to the problem
21:     S ⟵ schedule of jobs in S' completed after d_max prepended to S
22:     𝒯 ⟵ 𝒯 − all jobs in S' completed after d_max
23:     // Jobs that are rejected form a partial solution to the problem
24:     S̄ ⟵ S̄ + S̄'
25:     𝒯 ⟵ 𝒯 − S̄'
26:  }
27:  compress schedule S to remove idle time
28:  return S and S̄
```

Figure 7.4: An algorithm to solve $1||(\sum_S w_jT_j + \sum_{\bar{S}} e_j)$ using the FPAS from Section 7.5.2.

Every step forward in the world was formerly
made at the cost of mental and physical torture.

Nietzsche

Chapter 8

The Complexity of Scheduling with Separation Constraints

Those two Things had to stop.
Then I said to the cat,
"Now you do as I say.
You pack up those Things
And you take them away!"

Dr. Seuss, *The Cat in the Hat*

This chapter examines the complexity of scheduling problems that contain separation constraints, probing the boundary between easily solved problems and their more difficult generalizations. The scheduling with rejection problem formulation to solve the hardware-software partitioning problem uses separation constraints to model communication delays. Therefore, it is important to understand how the use of separation constraints affects the complexity of the scheduling with rejection problem. Recent work [8] has shown that most single machine scheduling problems containing separation constraints are at least as hard as the corresponding multiple machine scheduling problem without separation constraints. Although this relationship is not always true, it indicates that the addition of separation constraints can drastically increase the complexity of a single machine scheduling problem.

8.1 Introduction

A complexity hierarchy describing the relationships between scheduling problems that differ only in their objective functions is shown in Figure 7.1. This hierarchy indicates that the $C_{\max}$ and $\sum C_j$ objective functions are the simplest. Therefore, proving that a problem is $\mathcal{NP}$-hard for these two objective functions proves that it is $\mathcal{NP}$-hard for all objective functions in the complexity hierarchy.

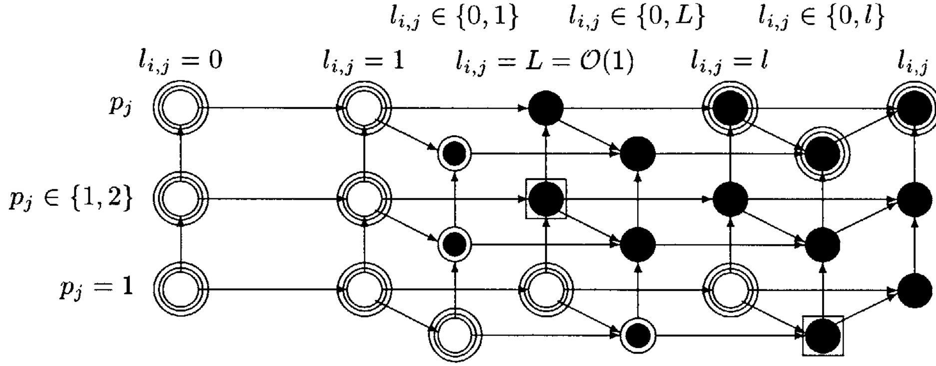
We examine the complexity of several single machine scheduling problems involving separation constraints for both the $C_{\max}$ and the $\sum C_j$ objective functions. We prove that most of these problems are $\mathcal{NP}$-complete in the strong sense[1] for both of these objective functions; therefore, the problems are $\mathcal{NP}$-complete in the strong sense for all of the standard objective functions. Table 8.1 summarizes my complexity results.

Problem	*Presented*
$^*1\|chain; pmtn; l_{i,j} = l\|C_{\max}$	Section 8.2.2
$^*1\|chain; pmtn; l_{i,j} = l\|\sum C_j$	Section 8.2.2
$^*1\|chain; p_j = 1; l_{i,j} \in \{0, l\}\|C_{\max}$	Section 8.2.3
$^*1\|chain; p_j = 1; l_{i,j} \in \{0, l\}\|\sum C_j$	Section 8.2.3
$^*1\|chain; p_j \in \{1, 2\}; l_{i,j} = L\ (L \geq 2)\|C_{\max}$	Section 8.2.4
$^*1\|chain; p_j \in \{1, 2\}; l_{i,j} = L\ (L \geq 2)\|\sum C_j$	Section 8.2.4
$^*Pm\|chain; p_j = 1\|\sum w_j C_j$	Section 8.2.5
$^*1\|prec; p_j = 1; l_{i,j} = l\|C_{\max}$	Section 8.3.1
$^*1\|prec; p_j = 1; l_{i,j} = l\|\sum C_j$	Section 8.3.1

Table 8.1: Complexity results presented in this chapter. A '*' indicates the problem to be strongly $\mathcal{NP}$-hard.

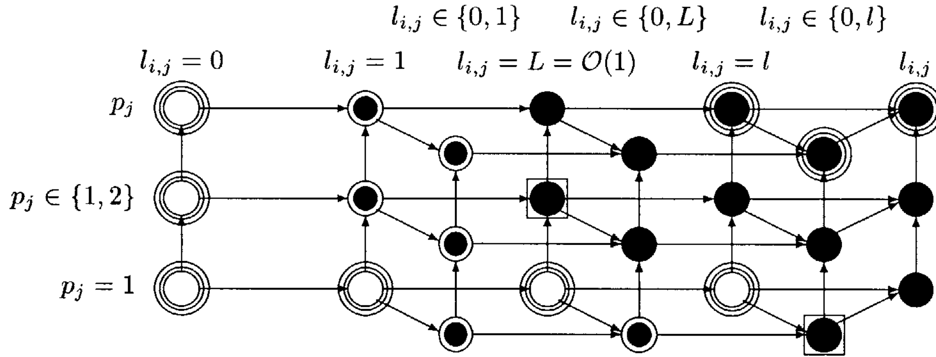
[1] A problem is $\mathcal{NP}$-complete in the strong sense if it cannot be solved by a pseudo-polynomial time algorithm. Thus, a strongly $\mathcal{NP}$-complete problem cannot be solved in pseudo-polynomial time even if its parameters are bounded by a polynomial.

$1|chain;\beta_2;\beta_3;\beta_5|C_{\max}$



(a)

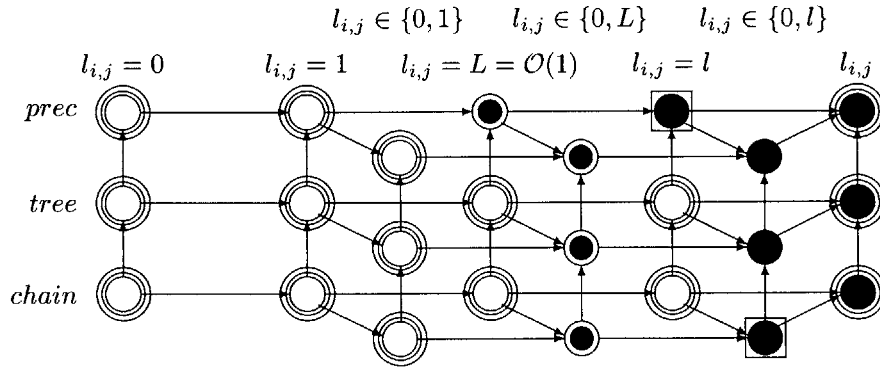
$1|chain;\beta_2;\beta_3;\beta_5|\sum C_j$



(b)

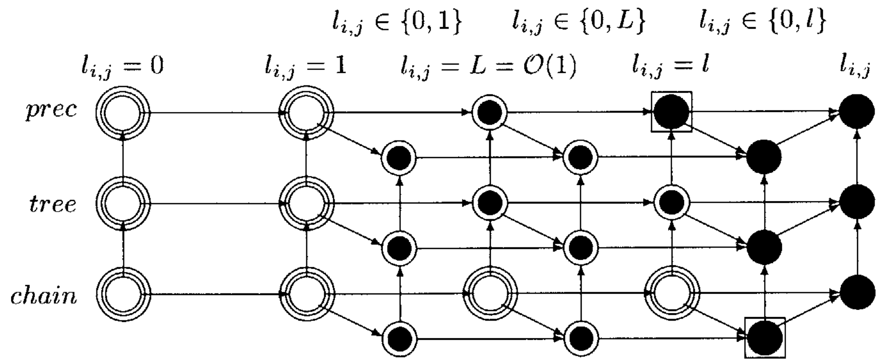
Figure 8.1: Graphical representation of the known boundary for single machine scheduling problems involving chain precedence constraints. Part (a) depicts the boundary for the makespan objective function. Part (b) depicts the boundary for the total completion time objective function. Nodes corresponding to problems with new complexity bounds presented in this dissertation are boxed. Nodes corresponding to problems whose complexity was previously known are doubly circled. We use the representation given by [24]. Problems are represented by circles, filled-in if known to be $\mathcal{NP}$-complete, empty if known to be in $\mathcal{P}$, and dotted if their complexity is unknown. An arrow from Π_1 to Π_2 signifies that Π_1 is a subproblem of Π_2.

$\mathbf{1}|\boldsymbol{\beta_1}; \boldsymbol{p_j} = \mathbf{1}; \boldsymbol{\beta_5}|\boldsymbol{C}_{\max}$



(a)

$\mathbf{1}|\boldsymbol{\beta_1}; \boldsymbol{p_j} = \mathbf{1}; \boldsymbol{\beta_5}|\sum \boldsymbol{C_j}$



(b)

Figure 8.2: Graphical representation of the known complexity boundary for unit execution time scheduling problems. Part (a) depicts the boundary for the makespan objective function. Part (b) depicts the boundary for the total completion time objective function.

My new complexity results narrow the boundary between known polynomial time solvable problems and known $\mathcal{NP}$-complete problems. Figure 8.1 graphically illustrates the boundary with regard to possible task processing times and distance constraints on a single machine. Figure 8.2 graphically illustrates the boundary with regard to possible constraint topologies and distance constraints on a single machine. Each circle represents a problem. A filled-in circle represents a known $\mathcal{NP}$-complete problem. An empty circle represents a known polynomial-time solvable problem, and a dotted circle represents an unknown complexity for the problem. Problems with new complexity results obtained in this dissertation are marked with a square. Problems whose complexity was previously known are marked with a circle.

8.2 Chain Structured Tasks

In this section, we examine the complexity of several problems involving chain structured tasks, i.e., problems in which the precedence constraints form sets of chains. I show that for the makespan, $C_{\max}$, and the total completion time, $\sum C_j$, objective functions these problems are strongly $\mathcal{NP}$-hard; therefore, they are strongly $\mathcal{NP}$-hard for all of the objective functions shown in Figure 7.1.

The 3-Partition problem is reduced to all of the various scheduling problems in their respective $\mathcal{NP}$-hardness proofs. The 3-Partition problem is stated as follows.

> Given a set of $3q$ elements $A = \{a_1, a_2, \dots, a_{3q}\}$, a bound B, and a size $s(a_j)$ for each $a_j \in A$ such that $B/4 < s(a_j) < B/2$ and $\sum_{a_j \in A} s(a_j) = qB$, can A be partitioned into q disjoint sets A_1, A_2, $\dots$, A_q such that for $1 \leq k \leq q$, $\sum_{a_j \in A_k} s(a_j) = B$?

For each scheduling problem Π in which this reduction is performed I show that for every instance of 3-Partition we can compute an instance of Π and a value z in polynomial or pseudo-polynomial time such that the instance of 3-Partition has a 'Yes' solution if and only if the instance of Π has a schedule of length at most z (for the makespan objective) or the sum of the completion times is at most z (for the

total completion time objective). The strong $\mathcal{NP}$-hardness results follow from the fact that 3-Partition is $\mathcal{NP}$-complete in the strong sense [24].

8.2.1 $\mathbf{1|chain; l_{j,k} = l|C_{max}, \sum C_j}$

The most basic chain structured tasks scheduling problems involving distance constraints are $1|chain; l_{j,k} = l|C_{\max}, \sum C_j$. The complexity of these problems was first examined by Balas et. al.[4] and Brucker and Knust[9] respectively. Balas et. al. were able to prove the strong $\mathcal{NP}$-completeness of $1|chain; l_{j,k} = l|C_{\max}$ with a simple reduction from 3-Partition. Brucker and Knust extended this proof in a straight forward manner to handle the total completion time objective function. The problems examined in the remainder of this section are restricted versions of these two problems (in the sense that they contain additional β constraints compared with these problems), and their respective $\mathcal{NP}$-completeness proofs are very similar in structure to the simple $\mathcal{NP}$-completeness proofs of $1|chain; l_{j,k} = l|C_{\max}, \sum C_j$. For this reason, I repeat the $\mathcal{NP}$-completeness proofs for $1|chain; l_{j,k} = l|C_{\max}, \sum C_j$.

Theorem 8.2.1 ([4]) *$1|chain; l_{j,k} = l|C_{\max}$ is $\mathcal{NP}$-complete in the strong sense.*

Proof This problem is clearly in $\mathcal{NP}$. To prove that it is strongly $\mathcal{NP}$-hard we reduce the 3-Partition Problem to it.

Given an instance of the 3-Partition Problem, we construct an instance of the $1|chain; l_{j,k} = l|C_{\max}$ problem as follows. There are $4q + 1$ tasks. For each $a_j \in A$, there is a task T_j with processing time $p_j = s(a_j)$. For each $i \in \{0, \ldots, q\}$, there is a task X_i with processing time $p_i = B$. The tasks X_i form a chain $X = X_0 \prec X_1 \prec \cdots \prec X_q$ with all distance constraints equal to B. The tasks T_j have no precedence constraints associated with them. We define $z = 2qB + B$. Note that the precedence constraints form a single chain and that the processing times of all the tasks is equal to z. Also, note that the chain X requires z time units to complete due to the distance constraints. Thus, any schedule that completes before the deadline, z, must not have any idle time, and the tasks

in X must be scheduled as soon as possible. The template formed by X is shown in Figure 8.3. It is easily verified that this reduction requires time polynomial in the parameters of the 3-Partition Problem.

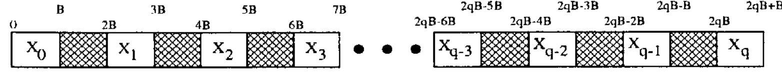


Figure 8.3: Template formed from the X chain in the proof of Theorem 8.2.1.

Suppose there exists a 'Yes' solution to the 3-Partition Problem. Since there exists a 'Yes' solution to the 3-Partition Problem, there exist q 3-element subsets S_i such that $\sum_{j \in S_i} p_j = B$. Since the chain X can be scheduled in time $2qB + B$ leaving q intervals each containing B time units, each of the q sets S_i may be scheduled within one interval for a total schedule time of $2qB + B$.

Suppose that a schedule of length z exists. The chain X requires a minimum time of $2qB + B$ to complete. Since there are exactly B time units between each of the tasks in X in a schedule that completes in time $2qB + B$ and preemption is not allowed, each set of tasks S_i scheduled between tasks X_{i-1} and X_i must have exactly B units of execution time. Furthermore, since $B/4 < p_j < B/2 \; \forall T_j$ there must be at least three tasks in each set S_i. The fact that there are exactly q sets S_i requires that there be exactly three tasks in each set S_i. Thus, there exists a partition into q 3-element subsets S_i, such that $\sum_{T_j \in S_i} p_j = \sum_{T_j \in S_i} s(a_j) = B$ $(1 \leq i \leq q)$, and the schedule is a witness to such a partition. Thus, we have a 'Yes' instance of 3-Partition. ∎

The reduction for the total completion time objective function is essentially identical to the reduction for the makespan objective function. The main difference is that the *template chain* X has a large number of extra tasks in it. Combined with the appropriate cost z, these extra tasks force the tasks in chain X to be scheduled as soon as possible. A large number of extra tasks are required in X since the exact

finish times of the tasks corresponding to the elements of A are not known. The sum of the finish times of the tasks corresponding to the elements in A may only be bounded. Thus, without the additional tasks in chain X, it is possible to have a solution with total completion time at most z when the corresponding 3-Partition problem does not contain a 'Yes' solution.

Theorem 8.2.2 **([9])** $1|chain; l_{j,k} = l|\sum C_j$ *is* $\mathcal{NP}$*-complete in the strong sense.*

Proof This problem is clearly in $\mathcal{NP}$. To prove that it is strongly $\mathcal{NP}$-hard we reduce the 3-Partition Problem to it.

Given an instance of the 3-Partition Problem, we construct an instance of $1|chain; l_{j,k} = l|\sum C_j$ as follows. There are $n = 3q + M$ tasks where $M = 2(3qB + B)q(q + 1) + q$. For each $a_j \in A$, there is a task T_j with processing time $p_j = s(a_j)$. For each $i \in \{1, \dots, M\}$, there is a task X_i with processing time $p_i = 3qB$. The tasks X_i form a chain $X = X_1 \prec X_2 \prec \cdots \prec X_M$ with all distance constraints equal to B. The tasks T_j have no precedence constraints associated with them. We define

$$z = \sum_{i=1}^{M}[3qBi + B(i-1)] + \sum_{j=1}^{q}[3j(3qB + B)].$$

Suppose that there exists a 'Yes' solution to the 3-Partition Problem. Since there exists a 'Yes' solution to the 3-Partition Problem, there exist q 3-element subsets S_i such that $\sum_{T_j \in S_i} p_j = B$. Since the chain X can be scheduled such that there exists $M - 1$ 'holes' each containing B time units, the tasks in X are scheduled as soon as possible. Thus, each of the q sets S_i may be scheduled within one hole. We schedule the sets S_i in the first q holes. The resulting mean flow time of the schedule satisfies

$$\begin{aligned}
\textstyle\sum_{j=1}^{n} C_j &= \textstyle\sum_{j=1}^{M} C_j + \sum_{i=1}^{3q} C_{M+i} \\
&\leq \textstyle\sum_{i=1}^{M}[3qBi + B(i-1)] + \sum_{j=1}^{q}[3j(3qB + b)] \\
&= z.
\end{aligned}$$

Suppose that a schedule with mean flow time at most z exists. The first q tasks of chain X must be scheduled as early as possible. Assume that this is not the case. Then,

$$\begin{aligned}
\textstyle\sum_{j=1}^{n} C_j &\geq \textstyle\sum_{j=1}^{M} C_j \\
&\geq \textstyle\sum_{j=1}^{q}[j3qB + (j-1)B] + \sum_{j=q+1}^{M}[j3qB + (j-1)B + 1] \\
&= z + (M-q) - \textstyle\sum_{j=1}^{q}[3(3qB+B)j \\
&\geq z + (M-q) - \tfrac{3}{2}(3qB+B)q(q+1) \\
&= z + (M-q) - \tfrac{3}{4}(M-q) \\
&> z,
\end{aligned}$$

which is a contradiction. Thus, the tasks of the chain X must be scheduled as early as possible.

In order for the mean flow time of the schedule to be at most z the tasks T_j, $1 \leq j \leq 3q$, must be scheduled before task X_{q+1}. Since there are exactly B time units between each of the tasks in X and preemption is not allowed, each set of tasks S_i scheduled between tasks X_{i-1} and T_i, $2 \leq i \leq q+1$, must have exactly B units of execution time. Furthermore, since $B/4 < p_j < B/2$ $\forall T_j$ there must be at least three tasks in each set S_i. The fact that there are exactly q sets S_i requires that there be exactly three tasks in each set S_i. Thus, there exists a partition into q 3-element subsets S_i, such that $\sum_{j\in S_i} p_j = \sum_{j \in S_i} s(a_j) = B$ $(1 \leq j \leq q)$, and the schedule is witness to such a partition. Thus, we have a 'Yes' instance of 3-Partition. ∎

8.2.2 $1|chain; pmnt; l_{j,k} = l|C_{max}, \sum C_j$

A slightly 'easier' version of the above problems allows preemption in a feasible schedule. The $\mathcal{NP}$-completeness proofs for the non-preemptive problems do not hold when preemption is allowed. Therefore, we must reexamine their complexity in light of preemption.

Balas et. al.[4] were able to prove that when preemption is allowed and the dis-

tance constraints are restricted to a set of two values that are inputs to the problem, the problem with the makespan objective function is strongly $\mathcal{NP}$-complete. I improve upon this result by showing that preemption does not improve the complexity of the problem when all distance constraints are equal to the same value, i.e., $1|chain; pmnt; l_{j,k} = l|C_{\max}, \sum C_j$ are strongly $\mathcal{NP}$-complete.

The $\mathcal{NP}$-completeness proofs are very similar to those for the non-preemptive problems. The main differences are that each element of A corresponds to a chain of tasks instead of a single task and the template chains form $2q$ intervals within which these tasks may be scheduled. The key idea is that the schedule from the first q intervals (corresponding roughly to the schedule in the non-preemptive proofs) constrains the possible schedules in the second q intervals (the mirror schedule).

Theorem 8.2.3 $1|chain; pmnt; l_{j,k} = l|C_{\max}$ *is* $\mathcal{NP}$*-complete in the strong sense.*

Proof $1|chain; pmnt; l_{j,k} = l|C_{\max}$ is clearly in $\mathcal{NP}$. To prove that it is also strongly $\mathcal{NP}$-hard, we reduce the 3-Partition Problem to it.

Without loss of generality we assume that $B \gg 3$ and that $s(a_j) \gg 3$, $\forall\, a_j \in A$. If this is not the case, then we may scale B and $s(a_j)$, $\forall\, a_j \in A$, by a constant without affecting the 3-Partition Problem. This constraint is required to ensure the validity of the transformation.

Given an instance of the 3-Partition Problem, we construct the following instance of $1|chain; pmnt; l_{j,k} = l|C_{\max}$. There are $3q^2 + 7q + 1$ tasks arranged in $3q + 2$ chains. For each $a_j \in A$ there is one chain C_j containing $q + 1$ tasks, $C_{(j,1)}$, $C_{(j,2)}$, $\ldots$, $C_{(j,(q+1))}$. The processing times for $C_{(j,1)}$ and $C_{(j,(q+1))}$ are $s(a_j)$. All other processing times for the tasks in C_j are equal to 1.

Two additional chains, X and Y, are created. X contains $2q + 1$ tasks each with processing time l (defined below). Y contains $2q$ tasks with processing times of

$$p(Y_i) = \begin{cases} 3q - 3i + 1 \text{ for } 1 \leq i \leq q \\ 3i - 3q - 2 \text{ for } q + 1 \leq i \leq 2q \end{cases}$$

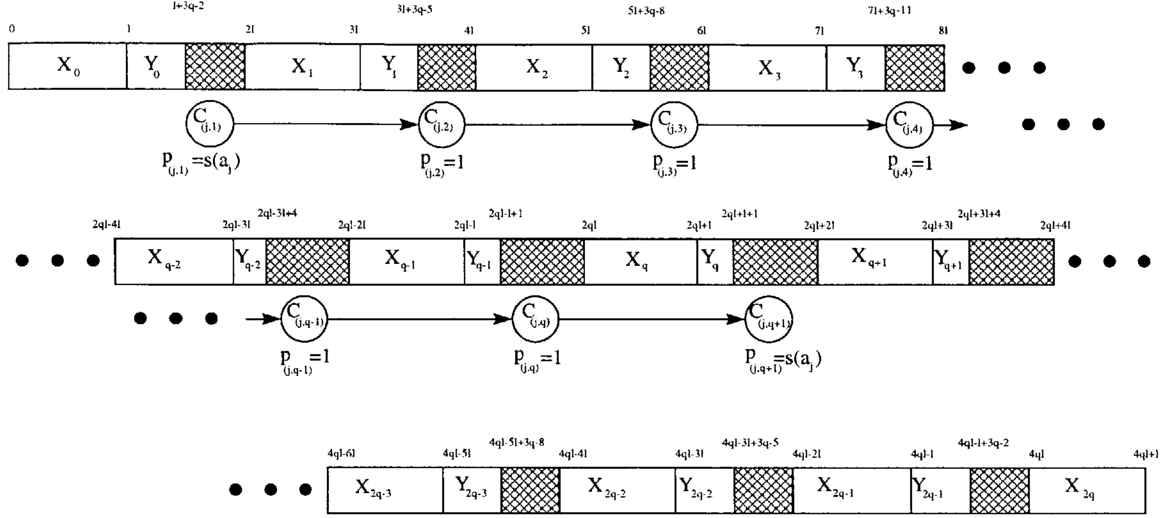


Figure 8.4: Template formed from the X chain and the Y chain in the proof of Theorem 8.2.3. This template limits the scheduling structure of the C_j chains.

All distance constraints are equal. We define the non-zero distance constraint to be

$$l = B + 3q - 2.$$

The deadline for the schedule is $z = 4ql + l$. Note that the processing times of all the tasks is equal to $4ql + l$; thus, any schedule that completes before the deadline must not have any idle time. Figure 8.4 illustrates how the template chain X and the enforcer chain Y create a template within which the C_j chains must be scheduled. It is easy to verify that this reduction requires time polynomial in the parameters of the 3-Partition problem.

Suppose we have a 'Yes' instance of 3-Partition. A schedule of length z is constructed as follows. Start the tasks of chain X as soon as possible. $X_{(2q+1)}$ finishes at time z. We have $2q$ intervals, $I_1, I_2, \ldots, I_{2q}$, each of length l within which the remaining tasks must be scheduled. Chain Y has $2q$ tasks; therefore, one task of chain Y must be executed during each interval. Furthermore, the distance constraints require that no more than one task of Y may be executed during any one interval. Thus, schedule

Y_i in interval I_i. After scheduling chain Y, the empty time slots in each interval I_i are characterized by the following function.

$$empty(I_i) = \begin{cases} B + 3(i-1) & 1 \leq i \leq q \\ B + 3(2q - i) & q+1 \leq i \leq 2q \end{cases}$$

By assumption of a 'Yes' instance of 3-Partition, there exists q disjoint 3-task sets, $H_1, H_2, \ldots, H_q$, with processing time of B and comprised of the first task from the chains C_j, $1 \leq j \leq 3q$. Schedule H_k in interval $I_k, 1 \leq k \leq q$. Consider task $C_{(j,1)}$ scheduled in interval I_i ($i \leq q$ due to how we scheduled the sets H_k). The tasks $C_{(j,2)}, C_{(j,3)}, \ldots, C_{(j,q)}$ can be scheduled during the next $q-1$ intervals. The additional empty time slots left by chain Y allow these 'pass-through' tasks to be scheduled in these intervals. By scheduling them thus, $C_{(j,(q+1))}$ can be scheduled in the interval I_{i+q}. The resulting schedule is feasible with a makespan of z.

Conversely, suppose that we have a schedule of length z. As before, the tasks in chain X must be scheduled as soon as possible, and exactly one task from chain Y must be scheduled in each interval I_i, $1 \leq i \leq 2q$. The tasks from the C_j chains are scheduled without idle times in the remaining time slots of the intervals. Furthermore, due to the distance constraints and lengths of the chains C_j, the first tasks from these chains, $C_{(j,1)}$, $1 \leq j \leq 3q$, must be scheduled in the first q intervals I_i, $1 \leq i \leq q$, and the last tasks from these chains, $C_{(j,(q+1))}$, $1 \leq j \leq 3q$, must be scheduled in the last q intervals I_i, $q+1 \leq i \leq 2q$.

Consider the first interval I_1 and only the first tasks in the chains C_j. Let S_1 be the set of $C_{(j,1)}$ tasks that are started and finished in I_1. Task Y_1 requires $3q-2$ time units in I_1 leaving B time units to schedule other tasks. Assume that $\sum_{C_{(j,1)} \in S_1} p_j = B - c$ for some $c > 0$. Consider the interval I_{q+1}. Y_{q+1} has an execution time of 1. There are at most $3q$ unit execution time tasks from the chains C_j that may be scheduled in this interval. The only additional tasks that may be scheduled in this time

interval are the final tasks of the chains C_j whose first task was scheduled in the first interval I_1 such that $C_{(j,1)} \in S_1$. $|S_1| \leq 3$ by the constraints in the 3-Partition Problem, and c is characterized by the following function due to our assumptions on B and $s(a_j), a_j \in A$.

$$c \geq \begin{cases} B & \text{if } |S_1| = 0 \\ B/2 + 1 & \text{if } |S_1| = 1 \\ 2 & \text{if } |S_1| = 2 \\ 3 & \text{if } |S_1| = 3 \end{cases}$$

Therefore, there are $c - 3 + |S_1| > 0$ idle time slots in interval I_{q+1}. It follows that our assumption is incorrect, and $\sum_{C_{(j,1)} \in S_1} p_j = B$. An iterative application of this argument leads to the identification of sets S_i with $\sum_{C_{(j,1)} \in S_i} p_j = B$ for $1 \leq i \leq q$ and to the conclusion that we have a 'Yes' instance of 3-Partition. ■

As in the previous section, the proof for $1|chain; pmnt; l_{j,k} = l|\sum C_j$ follows the proof for $1|chain; pmnt; l_{j,k} = l|C_{\max}$. This is achieved by making the template chain X and the enforcer chain Y much longer than the chains corresponding to the elements of A. The increased lengths of X and Y ensures that they are scheduled as soon as possible except possibly at the end. The chains corresponding to the elements of A must then be scheduled without idle times in the first $2q$ intervals caused by X if there is a 'Yes' instance of 3-Partition. Scheduling a task from these chains after all tasks in X (or most tasks in X) causes the total completion time of the resultant schedule to be greater than the target value z.

Theorem 8.2.4 *$1|chain; pmnt; l_{j,k} = l|\sum C_j$ is $\mathcal{NP}$-complete in the strong sense.*

Proof $1|chain; pmnt; l_{j,k} = l|\sum C_j$ is clearly in $\mathcal{NP}$. To prove that it is also strongly $\mathcal{NP}$-hard, we reduce the 3-Partition Problem to it.

Without loss of generality we assume that $B \gg 3$ and that $s(a_j) \gg 3$, $\forall\ a_j \in A$. If this is not the case, then we may scale B and $s(a_j)$, $\forall\ a_j \in A$, by a constant without affecting the 3-Partition Problem. Given an

instance of the 3-Partition Problem, we construct the following instance of $1|chain;pmnt;l_{j,k} = l|\sum C_j$. There are $3q+2$ chains. For each $a_j \in A$ there is one chain C_j containing $q+1$ tasks, $C_{(j,1)}, C_{(j,2)}, \ldots, C_{(j,(q+1))}$. The processing times for $C_{(j,1)}$ and $C_{(j,(q+1))}$ are $s(a_j)$. All other processing times for the tasks in C_j are equal to 1.

Two additional chains, X and Y, are created. X contains l^5+2q+1 tasks each with processing time l. Y contains l^5+2q tasks with processing times of

$$p(Y_i) = \begin{cases} 3q-3i+1 & \text{for } 1 \leq i \leq q \\ 3i-3q-2 & \text{for } q+1 \leq i \leq 2q \\ l & \text{for } 2q+1 \leq i \leq l^5+2q \end{cases}$$

All distance constraints are equal. We define the non-zero distance constraint to be

$$l = B + 3q - 2.$$

The target mean flow time for the schedule is

$$z = \sum_{i=1}^{l^5+2q+1} [l(2i-1)] + \sum_{i=2q+1}^{l^5+2q} [2li] + \sum_{i=1}^{2q} [2lif(i)],$$

where $f(i)$ is defined as

$$f(i) = \begin{cases} 4 + [(3q-2)-(3q-3i+1)] & 1 \leq i \leq q \\ 4 + [(3q-2)-(3i-3q-2)] & q+1 \leq i \leq 2q \end{cases}$$

It is easy to verify that this reduction requires time polynomial in the parameters of the 3-Partition problem.

Suppose we have a 'Yes' instance of 3-Partition. A schedule with mean flow time $\leq z$ is constructed as follows. Start the tasks of chain X as soon as possible. Thus,

$$\sum_{i=1}^{l^5+2q+1} C_{X_i} = \sum_{i=1}^{l^5+2q+1} [l(2i-1)].$$

We have l^5+2q intervals, $I_1, I_2, \ldots, I_{l^5+2q}$, each of length l within which the remaining tasks must be scheduled. Chain Y has l^5+2q tasks; therefore,

one task of chain Y must be executed during each interval. Furthermore, the distance constraints require that no more than one task of Y may be executed during any one interval. Thus, schedule Y_i in interval I_i. It is easily seen that if the first $2q$ tasks of X and Y are not scheduled as above, then the schedule will have a mean flow time $> z$. After scheduling chain Y, the empty time slots in each interval I_i are characterized by the following function.

$$empty(I_i) = \begin{cases} B + 3(i-1) & 1 \leq i \leq q \\ B + 3(2q-i) & q+1 \leq i \leq 2q \\ 0 & 2q+1 \leq i \leq l^5 + 2q \end{cases}$$

By assumption of a 'Yes' instance of 3-Partition, there exists q disjoint 3-task sets, $H_1, H_2, \ldots, H_q$, with processing time of B and comprised of the first task from the chains C_j, $1 \leq j \leq 3q$. Schedule H_k in interval I_k, $1 \leq k \leq q$. Consider task $C_{(j,1)}$ scheduled in interval I_i ($i \leq q$ due to how we scheduled the sets H_k). The tasks $C_{(j,2)}$, $C_{(j,3)}$, $\ldots$, $C_{(j,q)}$ can be scheduled during the next $q-1$ intervals. The additional empty time slots left by chain Y allow these 'pass-through' tasks to be scheduled in these intervals. By scheduling them thus, $C_{(j,(q+1))}$ can be scheduled in the interval I_{i+q}. The resulting schedule is feasible with a mean flow time $\leq z$.

Conversely, suppose that we have a schedule with a mean flow time $\leq z$. As before, the tasks in chain X must be scheduled as soon as possible, and exactly one task from chain Y must be scheduled in each interval I_i, $1 \leq i \leq l^5 + 2q$. The tasks from the C_j chains are scheduled without idle times in the remaining time slots of the intervals. If this is not the case, then there must be at least one task, T_k, with completion time $C_k \geq (l^5 + 2q + 1)B \geq \sum_{i=1}^{2q}[2lif(i)]$ contradicting our assumption on the mean flow time of the schedule. Furthermore, due to the distance constraints and lengths of the chains C_j, the first tasks from these chains, $C_{(j,1)}$, $1 \leq j \leq 3q$, must be scheduled in the first q intervals I_i, $1 \leq i \leq q$, and the last tasks

from these chains, $C_{(j,(q+1))}$, $1 \le j \le 3q$, must be scheduled in the last q intervals I_i, $q+1 \le i \le 2q$.

Consider the first interval I_1 and only the first tasks in the chains C_j. Let S_1 be the set of $C_{(j,1)}$ tasks that are started and finished in I_1. Task Y_1 requires $3q-2$ time units in I_1 leaving B time units to schedule other tasks. Assume that $\sum_{C_{(j,1)} \in S_1} p_j = B - c$ for some $c > 0$. Consider the interval I_{q+1}. Y_{q+1} has an execution time of 1. There are at most $3q$ unit execution time tasks from the chains C_j that may be scheduled in this interval. The only additional tasks that may be scheduled in this time interval are the final tasks of the chains C_j whose first task was scheduled in the first interval I_1 such that $C_{(j,1)} \in S_1$. $|S_1| \le 3$ by the constraints in the 3-Partition Problem, and c is characterized by the following function due to our assumptions on B and $s(a_j), a_j \in A$.

$$c \ge \begin{cases} B & \text{if } |S_1| = 0 \\ B/2+1 & \text{if } |S_1| = 1 \\ 2 & \text{if } |S_1| = 2 \\ 3 & \text{if } |S_1| = 3 \end{cases}$$

Therefore, there are $c - 3 + |S_1| > 0$ idle time slots in interval I_{q+1}. It follows that our assumption is incorrect, and $\sum_{C_{(j,1)} \in S_1} p_j = B$. An iterative application of this argument leads to the identification of sets S_j with $\sum_{C_{(j,1)} \in S_i} p_j = B$ for $1 \le i \le q$ and to the conclusion that we have a 'Yes' instance of 3-Partition. ■

8.2.3 $\mathbf{1|chain; p_j = 1; l_{j,k} \in \{0, l\}|C_{max}, \sum C_j}$

The strong $\mathcal{NP}$-completeness results for the preemptive version of the problem suggests that if the problem is restricted to have integral processing times and distance constraints, then preemptions may occur at integral boundaries only without affecting the complexity of the problem. The following lemma formalizes this observation.

Lemma 8.2.1 *If all input parameters are integral valued, then there exists an optimal solution to* $1|\beta_1; pmtn; \beta_3; \beta_5 \mid C_{\max}, \sum C_j$ *such that all preemptions occur at integral time boundaries.*

Proof We will prove this lemma by constructing an optimal solution S' containing preemptions only at integral time boundaries from an optimal solution S that may have preemptions at non-integral time boundaries. The solution S' will be constructed by iteratively 'fixing' a unit of execution time from a single task into a time slot. Once a unit of execution time is 'fixed' at a particular time slot it will remain at that time slot for the remainder of the iterations.

Let S be an optimal solution to a problem in $1|\beta_1; pmtn; \beta_3; \beta_5 \mid C_{\max}, \sum C_j$. Let t_i be the first time slot in S that has more than one task scheduled in it. Fix the schedule through time slot t_{i-1}. Considering only non-fixed task segments, let task C be the first task to complete an integral amount of processing after time t_i in S. Let $t_j > t_i$ be the time slot in which this unit of C completes. We may move this unit of processing time of C to time slot t_k where $t_i \leq t_k \leq t_j$ and t_k is the earliest time slot not already containing a fixed unit of processing time in which this unit of C may be scheduled. Fix this unit of C in time slot t_k. Note that due to distance constraints t_k may not be equal to t_i.

All non C tasks that were scheduled in the interval $[t_k, t_j]$ in S may be 'compressed' into the interval $[t_k + 1, t_j]$ where the compression does not change the scheduled time of fixed task segments. The compression maintains the ordering of the compressed tasks in the schedule, and it does not change the completion times of any task in the schedule S except possibly decreasing the completion time of task C. Thus, the modified schedule is still optimal.

By iteratively finding a time slot t_i and a task C, we construct the schedule S' that contains preemptions only at integral boundaries, and the

lemma is proven. ∎

The complexity results for the makespan scheduling problem containing chains, unit execution time tasks, and all distance constraints equal to either zero of l for some l input to the problem follow directly from Theorem 8.2.3 and Lemma 8.2.1.

Theorem 8.2.5 $1|chain; p_j = 1; l_{j,k} \in \{0, l\}|C_{\max}$ *is $\mathcal{NP}$-complete in the strong sense.*

Proof Follows from Theorem 8.2.3 and Lemma 8.2.1. ∎

Likewise, the complexity for the mean weighted flow time objective follows from Theorem 8.2.4 and Lemma 8.2.1.

Theorem 8.2.6 $1|chain; p_j = 1; l_{j,k} \in \{0, l\} \mid \sum w_j C_j$ *is $\mathcal{NP}$-complete in the strong sense.*

Proof Follows from Theorem 8.2.4 and Lemma 8.2.1. ∎

However, we can do better than this. The complexity for the total completion time objective function follows from the complexity for the makespan when all processing times are equal to one.

Theorem 8.2.7 $1|chain; p_j = 1; l_{j,k} \in \{0, l\}|\sum C_j$ *is $\mathcal{NP}$-complete in the strong sense.*

Proof We restrict the problem to instances where the optimum schedule S_{opt} contains no idle time. The sum of the completion times for S_{opt} is $\sum_{i=1}^{C_{\max}} i$ $= \sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$. Solving for the optimum sum of completion times is equivalent to solving for the optimum makespan. The theorem then follows from Theorem 8.2.5. ∎

8.2.4 $1|\mathbf{chain}; \mathbf{p_j} \in \{1,2\}; \mathbf{l_{j,k}} = \mathbf{L}\ (\mathbf{L} \geq 2)|\mathbf{C_{max}}, \sum \mathbf{C_j}$

We now turn our attention to the case when the values of the distance constraints are no longer part of the input to the problem, but instead are a fixed value. We consider the problem when all distance constraints are equal to a fixed value L. When $L = 0$, $1|\beta_1; \beta_2; \beta_3; l_{i,j} = 0|C_{\max}, \sum C_j$ are solvable in polynomial time. When $L = 1$, $1|\beta_1; \beta_2; \beta_3; l_{i,j} = 1|C_{\max}$ are solvable in polynomial time; however, the complexity of many of these problems is unknown for the sum of completion time objective function $\sum C_j$. We consider problems for which $L \geq 2$ and $\beta_1 = chain$. We further restrict the problem to contain only tasks with processing times of either 1 or 2. In other words, we examine the complexity for $1|chain; p_j \in \{1,2\}; l_{j,k} = L\ (L \geq 2)|C_{\max}, \sum C_j$. I prove that $1|chain; p_j \in \{1,2\}; l_{j,k} = L\ (L \geq 2)|C_{\max}, \sum C_j$ are strongly $\mathcal{NP}$-complete problems. These results are stronger than the results from Section 8.2.1.

Theorem 8.2.8 *$1|chain; p_j \in \{1,2\}; l_{j,k} = L\ (L \geq 2)|C_{\max}$ is $\mathcal{NP}$-complete in the strong sense.*

Proof This problem is clearly in $\mathcal{NP}$. To prove that it is strongly $\mathcal{NP}$-hard we reduce the 3-Partition Problem to it.

Given an instance of the 3-Partition Problem, we construct an instance of the $1|chain; p_j \in \{1,2\}; l_{j,k} = L\ (L \geq 2)|C_{\max}$ problem as follows. There are $[(6+B)q+1](L-2)+3q(3+B)+1$ tasks. For each $a_j \in A$, there is a chain C_j consisting of $s(a_j)+1$ tasks, $C_{j,1} \prec C_{j,2} \prec \cdots \prec C_{j,s(a_j)+1}$, with processing times $p_{j,k} = 1$, $1 \leq k \leq s(a_j)$, and $p_{j,s(a_j)+1} = 2$.

L additional chains, X_i, $1 \leq i \leq L-1$, and Y, are created. X_i, $1 \leq i \leq L-1$, contains $(6+B)q+2$ tasks each with processing time 1. Y contains $(3+B)q+1$ tasks with processing times of

$$p(Y_i) = \begin{cases} 2 & i = 0 \\ 1 & j(3+B)+1 \leq i \leq j(3+B)+B, 0 \leq j \leq q-1 \\ 2 & j(3+B)+B+1 \leq i \leq j(3+B)+B+3, 0 \leq j \leq q-1. \end{cases}$$

All distance constraints are equal to a constant $L \geq 2$.

We define $z = [(6+B)q+1]L + L - 1$. Note that the sum of the processing times of all tasks is equal to z; therefore, any schedule that completes by time z must not have any idle time. Figure 8.5 illustrates how the template chains X_i and the enforcer chain Y create a template within which the C_j chains must be scheduled. It is easy to verify that this reduction requires time polynomial in the parameters of the 3-Partition problem.

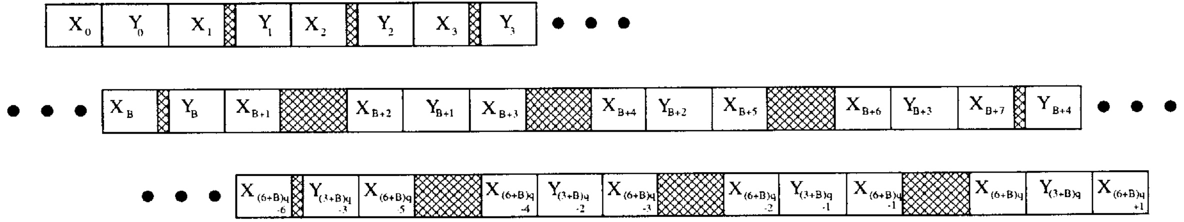


Figure 8.5: Template formed from the X_i chains and the Y chain in the proof of Theorem 8.2.8. X_k in the figure corresponds to the set of tasks $X_{i,k}$, $1 \leq i \leq L-1$. The shaded regions indicate unused processing times after chains X_i, $1 \leq i \leq L-1$, and Y have been scheduled.

Suppose that there exists a 'Yes' solution to the 3-Partition Problem. Schedule the tasks of chains X_i, $1 \leq i \leq L-1$, as soon as possible. $X_{i,(6+B)q+2}$ finish at times $[z-L+1, z]$. We have $(6+B)q+1$ intervals, I_1, I_2, ... , $I_{(6+B)q+1}$, each of length 2 within which the remaining tasks can be scheduled. Chain Y has $(3+B)q+1$ tasks; however, due to the processing times of the tasks in X_i and the distance constraints there are $3q$ intervals during which a task from Y cannot be scheduled, e.g., the interval between tasks $X_{i,B+6}$ and $X_{i,B+7}$ $\forall i$. Therefore, we must schedule the tasks of Y as early as possible. After scheduling chain Y the empty time slots in each

interval I_i are characterized by the following function

$$empty(I_i) = \begin{cases} 0 & i = \begin{cases} 1 \\ j(B+6)+B+3 & 0 \leq j \leq q-1 \\ j(B+6)+B+5 & 0 \leq j \leq q-1 \\ j(B+6)+B+7 & 0 \leq j \leq q-1 \end{cases} \\ \\ 1 & i = k + j(B+6)+1,\ 1 \leq k \leq B,\ 0 \leq j \leq q-1 \\ \\ 2 & i = \begin{cases} j(B+6)+B+2 & 0 \leq j \leq q-1 \\ j(B+6)+B+4 & 0 \leq j \leq q-1 \\ j(B+6)+B+6 & 0 \leq j \leq q-1 \end{cases} \end{cases}$$

By assumption of a 'Yes' instance of 3-Partition, there exists q disjoint 3-element sets, H_1, H_2, ... , H_q, with each element a_j corresponding to chain C_j. Schedule the corresponding chains of the elements in H_1 during the first $3 + B$ non-full intervals. Note that of these $3 + B$ intervals, each of the first B intervals has one idle time unit and each of the last three intervals has 2 idle time units. The tasks corresponding to H_1 consist of B tasks with $p_j = 1$ and three tasks with $p_j = 2$. The precedence constraints are such that the tasks corresponding to H_1 may be scheduled in the first $3 + B$ non-full intervals. (A non-full interval is an interval that contains a non-zero amount of unused processing time after all chains X_i and Y are scheduled.) Similarly, the tasks corresponding to H_2 may be scheduled in the next $3 + B$ non-full intervals, and so on. The resulting schedule is feasible and has a makespan of z.

Conversely, suppose that we have a schedule of length z. As before, the tasks in chains X_i, $1 \leq i \leq L-1$, must be scheduled as soon as possible, and the tasks from chain Y must be scheduled as soon as possible within the $(6 + B)q + 1$ intervals. The tasks from the C_j chains are scheduled without idle times in the remaining time slots of the schedule. Furthermore, tasks

with an execution time of 1 are only scheduled in intervals containing a task of Y with an execution time of 1, and tasks with an execution time of 2 are only scheduled in intervals that do not contain a task from Y.

Consider the first $3 + B$ non-full intervals. Each of the first B intervals contains a unit execution time task in a chain. By assumption on a schedule of length z, intervals I_{B+2}, I_{B+4}, and I_{B+6} each contain a non-Y task with execution time of 2. Since all non-Y tasks with an execution time of 2 are the final tasks in the C chains, there must be three chains C_i, C_j, and C_k, that have their respective first $s(a_i)$, $s(a_j)$, and $s(a_k)$, tasks scheduled in the first B intervals. Therefore, chains C_i, C_j, and C_k are completely scheduled during the first $3 + B$ non-full intervals. C_i, C_j, and C_k correspond to set H_1 containing the elements a_i, a_j, and a_k such that $|H_1| = s(a_i) + s(a_j) + s(a_k) = B$. An iterative application of this argument over each of the q sets of $3 + B$ non-full intervals leads to the identification of sets H_j with $|H_j| = B$ for $1 \leq j \leq q$ and to the conclusion that we have a 'Yes' instance of 3-Partition. ■

The complexity proof for the total completion time objective function is identical to the complexity proof for the makespan objective function. This is because the reduction forces the tasks to have a known sum of completion times. There is no variability in the sum of completion times if there is a 'Yes' solution to the 3-Partition problem. Therefore, by replacing z in the complexity proof for the makespan objective function with the sum of the completion times of chains $X_i, 1 \leq i \leq L-1$, Y, and C_j, $\forall\ a_j \in A$, the proof remains valid for the total completion time objective function.

Theorem 8.2.9 $1|chain; p_j \in \{1,2\}; l_{j,k} = L\ (L \geq 2)|\sum C_j$ *is $\mathcal{NP}$-complete in the strong sense.*

8.2.5 Complexity Boundary Analysis involving Chains

I have proven the strong $\mathcal{NP}$-completeness of several single machine problems involving chain structured tasks. By extension, the multiple machine versions of these prob-

lems are also strongly $\mathcal{NP}$-complete. These results lead naturally to the question of where is the boundary between polynomial time solvable problems and $\mathcal{NP}$-complete problems.

Table 8.2 presents the known boundary for the single machine case with respect to the makespan and total completion time objective functions. Polynomial time solvable problems are denoted by a 'p.' Strongly $\mathcal{NP}$-complete problems are denoted by a '$*$.' And, problems with an unknown complexity are denoted by a '?.' Note that the complexity results obtained in this section are all new minimal $\mathcal{NP}$-complete results.

A graphical representation of this boundary with respect to allowable task processing times and distance constraints is shown in Figure 8.1. The graph in Part (a) of the figure depicts the boundary for the makespan objective function, and the graph in Part (b) of the figure depicts the boundary for the total completion time objective function.

We now examine the complexity boundary involving multiple machine problems and problems that do not involve distance constraints. In order to more fully delineate the boundary I present one additional $\mathcal{NP}$-completeness proof for a parallel machine scheduling problem that does not involve distance constraints, namely $Pm|chain; p_i = 1|\sum w_j C_j$ for any fixed m with $m \geq 2$. Du, Leung, and Young proved that $Pm|chain; pmtn|\sum w_j C_j$ is strongly $\mathcal{NP}$-complete by showing that preemption can not improve the mean weighted flow time to $Pm|chain|\sum w_j C_j$ [19]. This result suggests that requiring all processing times to be equal to one time unit, i.e., β_3 is set to $p_j = 1$, does not reduce the complexity of the problem. I prove this to be the case after stating the main results obtained in [19].

Theorem 8.2.10 ([19]) *$Pm|chain|\sum w_j C_j$ is strongly $\mathcal{NP}$-complete.*

Theorem 8.2.11 ([19]) *Preemption cannot reduce the mean weighted flow time for a set of chains.*

Theorem 8.2.12 ([19]) *$Pm|chain; pmtn|\sum w_j C_j$ is strongly $\mathcal{NP}$-complete.*

p	$1\|chain; p_j = 1; l_{i,j} = l\|C_{\max}$	[43]
p	$1\|chain; p_j = 1; l_{i,j} = l\|\sum C_j$	[9]
p	$1\|chain; p_j = 1; l_{i,j} \in \{0,1\}\|C_{\max}$	[5]
?	$1\|chain; p_j = 1; l_{i,j} \in \{0,1\}\|\sum C_j$	
?	$1\|chain; p_j = 1; l_{i,j} \in \{0,L\}\ (L \geq 2)\|C_{\max}$	
?	$1\|chain; p_j = 1; l_{i,j} \in \{0,L\}\ (L \geq 2)\|\sum C_j$	
$*$	$1\|chain; p_j = 1; l_{i,j} \in \{0,l\}\|C_{\max}$	Section 8.2.3
$*$	$1\|chain; p_j = 1; l_{i,j} \in \{0,l\}\|\sum C_j$	Section 8.2.3

(a)

p	$1\|chain; l_{i,j} = 1\|C_{\max}$	[23]
?	$1\|chain; l_{i,j} = 1\|\sum C_j$	
?	$1\|chain; pmtn; l_{i,j} = L\ (L \geq 2)\|C_{\max}$	
?	$1\|chain; pmtn; l_{i,j} = L\ (L \geq 2)\|\sum C_j$	
$*$	$1\|chain; pmtn; l_{i,j} = l\|C_{\max}$	Section 8.2.2
$*$	$1\|chain; pmtn; l_{i,j} = l\|\sum C_j$	Section 8.2.2
$*$	$1\|chain; p_j \in \{1,2\}; l_{i,j} = L\ (L \geq 2)\|C_{\max}$	Section 8.2.4
$*$	$1\|chain; p_j \in \{1,2\}; l_{i,j} = L\ (L \geq 2)\|\sum C_j$	Section 8.2.4

(b)

Table 8.2: Complexity boundary involving single machine chain scheduling problems and the makespan and total completion time objective functions.

I now show that $Pm|\text{chain}; p_j = 1|\sum w_j C_j$ is strongly $\mathcal{NP}$-complete.

Theorem 8.2.13 *$Pm|chain; p_j = 1|\sum w_j C_j$ is strongly $\mathcal{NP}$-complete.*

Proof Consider the problem $Pm|\text{chain;pmtn}|\sum w_j C_j$. Without loss of generality we assume that all tasks in $Pm|\text{chain;pmtn}|\sum w_j C_j$ have integral processing times, p_j. We represent each task $T_j \in \mathcal{T}$ as a chain of p_j unit execution time tasks $C_{j,1} \prec \cdots \prec C_{j,p_j}$. To preserve the original precedence constraints, we add precedence constraints $C_{i,p_i} \prec C_{j,1}\ \forall\ T_i \prec T_j$ in the original problem.

With each task C_{j,p_j} we associate the weight w_j. The weight for all other tasks is set to zero.

This reduction is performed in pseudo-polynomial time since the number of tasks in the reduced problem is equal to $\sum p_j$.

Solving $Pm|\text{chain;}p_j = 1|\sum w_j C_j$ clearly finds a feasible schedule to $Pm|\text{chain;pmtn}|\sum w_j C_j$. By Theorem 8.2.11 the optimum schedule to $Pm|\text{chain;}p_j = 1|\sum w_j C_j$ is an optimum schedule to $Pm|\text{chain;pmtn}|\sum w_j C_j$. Therefore, solving $Pm|\text{chain;}p_j = 1|\sum w_j C_j$ solves $Pm|\text{chain;pmtn}|\sum w_j C_j$. Thus, $Pm|\text{chain;}p_j = 1|\sum w_j C_j$ is strongly $\mathcal{NP}$-complete. ■

Table 8.3 presents the known boundary involving multiple machine unit execution time chain scheduling problems and makespan, total completion time, and total weighted completion time objective functions.

8.3 Arbitrary Precedence Structured Tasks

We now examine the complexity of several problems involving `tree` and `prec` precedence constraint topologies. I show that for the makespan, $C_{\max}$, and total completion time, $\sum C_j$, objective functions the problems involving `prec` structured precedence constraint topologies are strongly $\mathcal{NP}$-hard; therefore, they are strongly $\mathcal{NP}$-hard for all of the objective functions shown in Figure 7.1.

p	$P\|chain; p_j = 1\|\sum C_j$	[31]
p	$P\|chain; p_j = 1; l_{i,j} = l\|C_{\max}$	[43]
p	$1\|chain; p_j = 1; l_{i,j} \in \{0,1\}\|C_{\max}$	[6]
p	$1\|chain; p_j = 1; \|\sum w_j C_j$	[42]
*	$1\|chain; p_j = 1; l_{i,j} \in \{0,l\}\|C_{\max}$	Section 8.2.3
*	$1\|chain; p_j = 1; l_{i,j} \in \{0,l\}\|\sum C_j$	Section 8.2.3
*	$1\|chain; p_j = 1; l_{i,j} = 1\|\sum w_j C_j$	[59]
*	$P2\|chain; p_j = 1\|\sum w_j C_j$	Section 8.2.5

Table 8.3: Complexity boundary involving multiple machine chain scheduling problems and the makespan and total (weighted) completion time objective functions.

8.3.1 $1|prec; p_j = 1; l_{j,k} = l|C_{\max}, \sum C_j$

We now consider problems with arbitrary precedence constraints, namely $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ and $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$. Unfortunately, allowing arbitrary precedence constraints makes the problems strongly $\mathcal{NP}$-hard.

I reduce the 3-Partition problem to $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ in a manner similar to the reductions used for chain structured tasks. Precedence constraints are used to create a specific template structure in a similar manner to the way the zero distance constraints and arbitrary execution times are used in the reductions involving chain structured tasks.

Theorem 8.3.1 $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ *is $\mathcal{NP}$-complete in the strong sense.*

Proof $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ is clearly in $\mathcal{NP}$. To prove that it is also strongly $\mathcal{NP}$-hard, we reduce the 3-Partition Problem to it.

Given an instance of the 3-Partition Problem, we construct the following instance of $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$. We define the non-zero distance constraint to be

$$l = B + 3q - 3.$$

There are $l(4q+1)+4q+2$ tasks arranged in $3q+1$ separate dags. For each $a_i \in A$ there is one dag D_i containing $2s(a_i)+q-1$ tasks, $D_{(i,1)}$, $D_{(i,2)}$, $\ldots$, $D_{(i,(2s(a_i)+q-1))}$. The precedence constraints within D_i are defined as follows. $D_{(i,j)} \prec D_{(i,s(a_i)+1)}$, $1 \leq j \leq s(a_i)$. $D_{(i,j)} \prec D_{(i,+1)}$, $s(a_i)+1 \leq j \leq s(a_i)+q-1$. $D_{(i,s(a_i)+q-1)} \prec D_{(i,j)}$, $s(a_i)+q \leq j \leq 2s(a_i)+q-1$.

One additional dag, X, is created. X contains $l(2q+1)+4q+2+2\sum_{i=1}^{q}(l-B-3(i-1))$ tasks. The central feature of X is a chain of $4q+2$ tasks, $X_1, X_2, \ldots, X_{4q+2}$. All other tasks of X are connected to at least one of these chain tasks and most are connected to two of the chain tasks. A non-chain task has precedence relations with chain tasks only. For simplicity in the definition of the remaining precedence constraints we note a non-chain task of X as Y_j. We also use the following invariant. If it is stated that chain task X_i precedes non-chain task Y_j, $X_i \prec Y_j$, then it is also true that Y_j precedes X_{i+3}, $Y_j \prec X_{i+3}$, if X_{i+3} exists.

The remaining precedence constraints in X are defined as follows. $Y_{0,j} \prec X_3$, $1 \leq j \leq l$. $X_i \prec Y_{i,j}$, $1 \leq j \leq l$, $1 \leq i \leq 4q$, and i is even. $X_i \prec Y_{i,j}$, $1 \leq j \leq l-B-3(o(i)-1)$, $1 \leq i \leq 2q$, i is odd, and $o(i)$ returns the index of i in the list of odd numbers greater than zero, e.g., $o(1)=1$, $o(3)=2$, $o(5)=3$, $o(7)=4$. $X_i \prec Y_{i,j}$, $1 \leq j \leq l-B-3(q-o(i-2q))$, $2q+1 \leq i \leq 4q$, i is odd, and $o(i)$ is defined as above.

All distance constraints are equal to l.

The deadline for the schedule is $z = l(4q+1)+4q+2$. Note that the processing times of all the tasks is equal to z; thus, any schedule that completes before the deadline must not have any idle time. It is easy to verify that this reduction requires time polynomial in the parameters of the 3-Partition problem.

Suppose we have a 'Yes' instance of 3-Partition. A schedule of length z is constructed as follows. Start the tasks of chain X as soon as possible. $X_{(4q+2)}$ finishes at time z. The remaining $Y_{i,j}$ tasks are constrained to

be scheduled in the time interval between task X_{i+1} and task X_{i+2}. The resultant template has $2q$ non-full intervals, $I_1, I_2, \ldots, I_{2q}$, within which the remaining tasks must be scheduled. These intervals occur between tasks X_i and X_{i+1} for $1 \leq i \leq 4q$ and i even. The number of empty time slots in each interval I_i are characterized by the following function.

$$empty(I_i) = \begin{cases} B + 3(i-1) & 1 \leq i \leq q \\ B + 3(2q - i) & q + 1 \leq i \leq 2q \end{cases}$$

By assumption of a 'Yes' instance of 3-Partition, there exists q disjoint B-task sets, $H_1, H_2, \ldots, H_q$, with processing time of B and comprised of the first $s(a_i)$ tasks from the dags D_i, $1 \leq i \leq 3q$. Schedule H_k in interval $I_k, 1 \leq k \leq q$. Consider tasks $D_{(i,1)}$, $D_{(i,2)}$, $\ldots$, $D_{(i,s(a_i))}$ scheduled in interval I_j ($j \leq q$ due to how we scheduled the sets H_k). The tasks $D_{(i,s(a_i)+1)}$, $D_{(i,s(a_i)+2)}$, $\ldots$, $D_{(i,s(a_i)+q-1)}$ can be scheduled during the next $q-1$ intervals. The additional empty time slots above the B needed to schedule H_j allow these 'pass-through' tasks to be scheduled in these intervals. By scheduling them thus, The tasks $D_{(i,s(a_i)+q)}$, $D_{(i,s(a_i)+q+1)}$, $\ldots$, $D_{(i,2s(a_i)+q-1)}$ can be scheduled in the interval I_{j+q}. The resulting schedule is feasible with a makespan of z.

Conversely, suppose that we have a schedule of length z. As before, the tasks in chain X must be scheduled as soon as possible, and the remaining $Y_{i,j}$ tasks are constrained to be scheduled in the time interval between task X_{i+1} and task X_{i+2}. The resultant template has $2q$ non-full intervals, I_1, I_2, $\ldots$, I_{2q}, within which the remaining tasks are scheduled. The tasks from the D_i dags are scheduled without idle times in the remaining time slots of the intervals. Furthermore, due to the distance constraints and lengths of the chains $D_{(i,s(a_i)+1)} \prec D_{(i,s(a_i)+2)} \prec \cdots \prec D_{(i,s(a_i)+q-1)}$, the first $s(a_i)$ tasks from these dags, D_i, $1 \leq i \leq 3q$, must be scheduled in the first q intervals I_j, $1 \leq j \leq q$, and the last $s(a_i)$ tasks from these dags must be scheduled in the last q intervals I_j, $q+1 \leq j \leq 2q$.

Consider the first interval I_1 and only the first $s(a_i)$ tasks in the dags D_i. Let S_1 be the set of dags D_i that have their first $s(a_i)$ tasks, $D_{(i,1)}$, $D_{(i,2)}, \ldots, D_{(i,s(a_i))}$, scheduled in I_1. The template dag leaves B time units to schedule other tasks in I_1. Assume that $\sum_{D_i \in S_1} s(a_i) = B - c$ for some $c > 0$. Consider the interval I_{q+1}. The template dag leaves $B + 3q - 3$ time units to schedule other tasks in I_{q+1}. There are at most $3q$ tasks, $D_{(i,j)}$, $s(a_i) + 1 \leq j \leq s(a_i) + q - 1$, from the dags D_i that may be scheduled in this interval. The only additional tasks that may be scheduled in this time interval are the final $s(a_i)$ tasks of the dags D_i whose first $s(a_i)$ tasks were scheduled in the first interval I_1 such that $D_i \in S_1$. $|S_1| \leq 3$ by the constraints in the 3-Partition Problem, and c is characterized by the following function due to our assumptions on B and $s(a_i)$, $a_i \in A$.

$$c \geq \begin{cases} B & \text{if } |S_1| = 0 \\ B/2 + 1 & \text{if } |S_1| = 1 \\ 2 & \text{if } |S_1| = 2 \\ 3 & \text{if } |S_1| = 3 \end{cases}$$

Therefore, there are $c - 3 + |S_1| > 0$ idle time slots in interval I_{q+1}. It follows that our assumption is incorrect, and $\sum_{D_i \in S_1} s(a_i) = B$. An iterative application of this argument leads to the identification of sets S_j with $\sum_{D_i \in S_j} s(a_i) = B$ for $1 \leq j \leq q$ and to the conclusion that we have a 'Yes' instance of 3-Partition. ■

We may assume without loss of generality that the optimum solution to $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ does not contain any idle times. Since this problem involves unit execution time tasks, the total completion time to the optimum schedule for $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ is $\sum C_j = \sum_{j=1}^{C_{\max}} j = \sum_{j=1}^{n} j$. This is the smallest total completion time that any schedule may have. Therefore, finding a schedule to $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$ with total completion time of $\sum_{j=1}^{n} j$ finds an optimum schedule to $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$, and $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$ is strongly $\mathcal{NP}$-complete. This result is formalized in the following theorem.

Theorem 8.3.2 $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$ *is $\mathcal{NP}$-complete in the strong sense.*

Proof We reduce the known strongly $\mathcal{NP}$-complete problem $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ to $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$. Without loss of generality we assume that the optimum schedule, S_{opt}, to $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ does not contain any idle time slots. The mean flow time for S_{opt} is $\frac{1}{n}\sum_{j=1}^{C_{\max}} j = \frac{1}{n}\sum_{j=1}^{n} j$.

Given an instance of $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$, we assign a weight of one to all tasks. Let $y = \sum_{j=1}^{n} j$. The optimum solution to $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$ is a solution to $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$ such that $\sum_{j=1}^{n} C_j \leq y$. Conversely, any solution to $1|prec; p_j = 1; l_{j,k} = l|\sum C_j$ such that $\sum_{j=1}^{n} C_j \leq y$ is clearly an optimum solution to $1|prec; p_j = 1; l_{j,k} = l|C_{\max}$. The theorem follows. ■

8.3.2 Complexity Boundary Analysis

In this section we have examined unit execution time problems involving non-chain structured tasks and constant distance constraints. Table 8.4 presents the known boundary for unit execution time scheduling problems involving various precedence constraint topologies with respect to the makespan and total completion time objective functions. Note that the results obtained in this section are either minimal $\mathcal{NP}$-complete results or maximal polynomial-time solvable results. Figure 8.2 depicts this boundary graphically.

p	$P\|intree; p_j = 1; l_{i,j} = l\|C_{\max}$	[43]
p	$P\|outtree; p_j = 1; l_{i,j} = l\|C_{\max}$	[10]
p	$1\|prec; p_j = 1; l_{i,j} \in \{0,1\}\ \|C_{\max}$	[6]
?	$1\|prec; p_j = 1; l_{i,j} \in \{0,1\}\ \|\sum C_j$	
?	$1\|prec; p_j = 1; l_{i,j} = L(L \geq 2)\ \|C_{\max}$	
?	$1\|prec; p_j = 1; l_{i,j} = L(L \geq 2)\ \|\sum C_j$	
*	$1\|prec; p_j = 1; l_{i,j} = l\|C_{\max}$	Section 8.3.1
*	$1\|prec; p_j = 1; l_{i,j} = l\|\sum C_j$	Section 8.3.1

(a)

p	$1\|prec; l_{i,j} = 1\|C_{\max}$	[23]
?	$1\|prec; l_{i,j} = 1\|\sum C_j$	
*	$1\|chain; p_j \in \{1,2\}; l_{i,j} = L\ (L \geq 2)\|C_{\max}$	Section 8.2.4
*	$1\|chain; p_j \in \{1,2\}; l_{i,j} = L\ (L \geq 2)\|\sum C_j$	Section 8.2.4

(b)

Table 8.4: Complexity boundary involving scheduling problems with the makespan and total completion time objective functions.

8.4 Approximation Bounds for $\mathbf{P|prec; p_j = 1; l_{i,j}|C_{\max}}$

As we have seen in the previous sections, most scheduling problems involving separation constraints are strongly $\mathcal{NP}$-hard even for chain structured tasks scheduled on a single machine. Therefore, it is very unlikely that a polynomial time or even a pseudopolynomial time algorithm will be found to solve these problems optimally. Due to this computational complexity, it is natural to ask if a near optimal solution can be found efficiently for any problem involving separation constraints. In this section, we answer this question in the positive by proving that an arbitrary list scheduling algorithm will find a solution that is within a factor of $2 - 1/(md+1)$ to the optimum for the problem $P|prec; p_j = 1; l_{i,j}|C_{\max}$. We also prove an approximation bound slightly better than $2 - 2/(md + m)$ for the Coffman-Graham algorithm applied to this problem.

8.4.1 Introduction

The relaxation of a problem from one of finding an optimum solution to one of finding an *approximately* optimum solution can significantly reduce the complexity of the problem. The primary difficulty with this relaxation is proving that the approximate solution is 'close enough' to the optimum solution. To ensure that 'close enough' is 'good enough,' the *error* of these *approximation algorithms* must be bounded. This bound tells how far from optimum a solution obtained using approximation algorithm $\mathcal{A}$ will be in the worst case. Ideally, we want to bound the absolute difference between the cost of a solution obtained from $\mathcal{A}$ on instance I of the problem and the optimum cost of the solution for instance I. Unfortunately, the *absolute approximation problems* for scheduling problems containing separation constraints are as difficult as their corresponding exact problems. Consequently, we concentrate on finding α-*approximation algorithms* for scheduling problems containing separation constraints. An α-approximation algorithm is a polynomial-time algorithm that returns a solution

with objective value at most α times the optimal value. α is often referred to as the worst-case performance guarantee or ratio of the algorithm.

We consider a general class of scheduling problems on identical parallel machines. We have a set $\mathcal{T}$ of n unit execution time (UET) tasks and m identical parallel machines. A set of precedence constraints $\prec$ is associated with $\mathcal{T}$. These precedence constraints force a partial ordering on any feasible schedule. We associate with every precedence constrained task pair $(i, j) \in \prec$ a nonnegative separation constraint $l_{i,j}$. This separation constraint requires that task j cannot begin execution until $l_{i,j}$ time units after task i has completed execution. The set of tasks is to be nonpreemptively scheduled onto the machines such that all precedence and separation constraints are met and the objective function is minimized. We only consider the makespan objective function in this paper. Using the common notation introduced in [28], we denote the most general of these problems as $P|prec; p_j = 1; l_{i,j}|C_{\max}$. We denote the maximum separation constraint in a problem instance by $d = \max_{(i,j) \in \prec} l_{i,j}$.

List scheduling algorithms, first analyzed by Graham [26] for scheduling problems without separation constraints, are among the simplest and most commonly used approximate solution methods for parallel machine scheduling problems [50]. These algorithms assign each task a priority by ordering all tasks in a list. A greedy algorithm then schedules the tasks as soon as they are ready, giving priority to those tasks that appear earliest in the list.

Bernstein, Rodeh, and Gertner [6] examined the approximation bound for list schedules applied to the single machine problem $1|prec; p_j = 1; l_{i,j}|C_{\max}$. They determined that an arbitrary list schedule has an approximation ratio of $2 - 1/(d + 1)$. Palem and Simons [53] extended this result to the identical parallel machines scheduling problem and determined that an arbitrary list schedule has an approximation ratio of $2 - 1/(md + m)$. In Section 8.4.2 we improve this approximation ratio to $2 - 1/(md + 1)$.

Since an arbitrary list can generate a schedule that is within a factor of two of the optimum, it is natural to ask whether or not a list generated by a particular algorithm can do any better. In Section 8.4.3 we analyze the Coffman-Graham algorithm [13]

for the identical parallel machine problem. Bernstein, Rodeh, and Gertner [6] determined that the approximation bound for this algorithm applied to the single machine problem is $2 - 2/(d + 1)$. We extend this result to the identical parallel machines problem proving a bound slightly better than $2 - 2/(md + m)$.

8.4.2 List Schedules

Definition of List Schedules

List schedules are a class of scheduling algorithms that form a schedule by greedily scheduling a list of tasks in the order the tasks are given in the list. The list scheduling algorithm is simply stated as follows: Given a *priority list* $\mathcal{L}$ of the tasks of $\mathcal{T}$ the list schedule S_L can be constructed by the following procedure:

1. Iteratively schedule the elements of S_L starting in time slot 1 such that during the i-th step, $\mathcal{L}$ is scanned from left to right and the first m ready tasks not yet scheduled are chosen to be executed during time slot i.

2. If less than m tasks are ready, then NOPs are inserted into S for the idle machines in time slot i.

The list scheduling algorithm is readily adapted to scheduling with distance constraints by augmenting the elements of the priority list to contain the initial ready time of the tasks. This is a function that will be updated during the execution of the list scheduling algorithm to ensure that no task is scheduled in violation of any distance constraints.

Consider a class of optimum schedules for UET scheduling. Since all the tasks in $\mathcal{T}$ have unit execution times, an optimum schedule does not need to leave a machine idle whenever a ready task exists. Therefore, an optimal schedule can always be found among list schedules. The following lemma formalizes this statement.

Lemma 8.4.1 *An optimum schedule may be constructed such that no machine is left idle during a time slot i if there are ready tasks during that time slot.*

Proof We show how to construct the desired optimum schedule from an arbitrary optimum schedule. Consider the optimum schedule S_O that contains idle machines during time slots in which tasks are ready to execute. Let time slot t_i be the earliest such time slot, and let m_a be an idle machine during that time slot. Let task T_k be a task ready during time slot t_i but scheduled at time slot $t_j > t_i$ on machine m_b. Task T_k may be rescheduled to time slot t_i on machine m_a without violating any constraints. Doing so creates another idle machine, m_b, during time slot t_j. Repeatedly performing this operation yields a schedule containing no idle machines during time slots in which there are ready tasks. ■

Since a list schedule will generate an optimal schedule given the optimal priority list L, the key to the success of the list scheduling algorithm is generating the optimal priority list. A list schedule requires time $\mathcal{O}(n+e)$ to schedule a priority list where n is the number of tasks and e is the number of precedence constraints. Therefore, generating the optimal priority list is as difficult as finding the optimal schedule. In the following section, we bound how far from optimal a list schedule can be, given a non-optimal priority list.

Approximation Bounds for Arbitrary List Schedules

We use the following lemma to determine the upper bound for the ratio $R = \frac{\omega(S_L)}{\omega(S_O)}$, where $\omega(S_L)$ is the makespan for an arbitrary list schedule S_L and $\omega(S_O)$ is the makespan for an optimal list schedule S_O. This lemma and much of the following analysis closely follows the analysis for distance constrained scheduling on one processor in [6].

Lemma 8.4.2 *Let T be a set of n tasks, and let $\omega(S_O) = \frac{n+k}{m}$, i.e., there are $k \geq 0$ NOPs in the optimum schedule on m processors. Let S_L be a list schedule of T such that $R = \frac{\omega(S_L)}{\omega(S_O)}$ is maximal. If $k \geq m$, then there exists a task system T' whose optimal schedules contain no NOPs and a list schedule S'_L for it such that $R' = \frac{\omega(S_L)}{\omega(S_O)} > R$.*

Proof We show how to construct T' from T. Without loss of generality we assume n is a multiple of m. If n is not a multiple of m, independent tasks may be added to T without affecting the optimum solution. Let T' consist of all tasks in T plus k independent tasks. An optimum schedule S'_O of T' can be obtained from the optimum schedule S_O of T by replacing the k NOPs in S_O with the k independent tasks from T'. Therefore, $\omega(S'_O) = \omega(S_O)$. S'_L is constructed by scheduling the k independent tasks first. The remaining schedule is identical to S_L. Therefore, $\omega(S'_L) = \omega(S_L) + k/m$. Thus, $R' = \frac{\omega(S'_L)}{\omega(S'_O)} = \frac{\omega(S_L)+k/m}{\omega(S_O)} > \frac{\omega(S_L)}{\omega(S_O)} = R$. ■

Let S_L be a list schedule of T, and assume that an idle processor exists at time slot t_i. The idle processor is *induced* by T_a and T_b if:

1. scheduled time slot of $T_a < t_i <$ scheduled time slot of T_b.

2. $(T_a, T_b) \in \prec$.

Note that every NOP of a list schedule must be induced by at least one pair of tasks. Let $C = C_1, \ldots, C_z$ be a directed path (chain) in T. C *covers* a NOP of S_L if the NOP is induced by a pair of tasks of C.

Lemma 8.4.3 ([6]) *Let T be a task system, and let S_L be a list schedule of T. Then there exists a directed path $C = C_1, \ldots, C_z$ in T that covers all the NOPs of S_L.*

Proof The proof given in [6] is valid for multiple processor systems. ■

We let the distance constraints be arbitrary but bounded from above. The maximum distance constraint in the system is d.

Theorem 8.4.1 *Let $(\mathcal{T}, \prec)$ be a task system with n tasks, S_O an optimal schedule and S_L an arbitrary list schedule of T. Then $R = \frac{\omega(S_L)}{\omega(S_O)} \leq 2 - \frac{1}{(1+md)}$, where m is the number of processors and d is the maximum distance constraint in $\prec$.*

Proof By Lemma 8.4.2 we may assume that S_O has no NOPs. Therefore, $\omega(S_O) = n/m$. Let k be the number of NOPs in S_L. Thus, $\omega(S_L) = (n+k)/m$ and $R = 1 + k/n$. By Lemma 8.4.3 there exists a directed path $C = C_1, \ldots, C_z$ that covers all the NOPs in S_L. Therefore, $\sum_{i=1}^{z-1} D(C_i) \geq k/r$, where $D(C_i)$ is the distance constraint from task C_i to task C_{i+1}. Notice that since $D(T_j) \leq d$ for all T_j, we have $z - 1 \geq k/(md)$. Since S_O has no NOPs, there must be k tasks in S_O that fill in the delays of C in S_L. Therefore, we get $n \geq k + z \geq k + k/md + 1$. Thus, $R = 1 + \frac{k}{n} \leq 1 + \frac{k}{(k+k/md+1)} = 2 - \frac{1}{(md/(1+md/k))} \leq 2 - \frac{1}{(1+md)}$. ■

This bound is valid for $md \geq 1$. Notice that when $m = 1$, this is the tight bound for distance constrained scheduling on one processor found by Bernstein [6]. However, for $d = 0$, Graham [27] reports the tight approximation bound to be $2 - \frac{1}{m-1}$ for $m > 2$.

8.4.3 The Coffman-Graham Algorithm

The CG Algorithm

Coffman and Graham [13] give a list scheduling algorithm that prioritizes the tasks based on their level with the tasks with the highest level having the highest priority. The level of a task is defined to be the length of the longest path from that task to a sink task in $\mathcal{T}$. However, instead of making an arbitrary choice among tasks with the same level, the Coffman-Graham priority rule makes a judicious choice of which task to give a higher priority.

The Coffman-Graham (CG) priority rule assigns each task $j \in \mathcal{T}$ a priority label $\alpha(j) \in \{1, 2, \ldots, n\}$. The algorithm to generate the priority list is given below.

1. Choose an arbitrary task j such that it does not have any successors, and define $\alpha(j) = 1$.

2. Suppose, for some $i \leq n$, that labels $1, 2, \ldots, i-1$ have been assigned. Let $\mathcal{R}$ be the set of tasks with no unlabelled successor. Let j^* be a task in $\mathcal{R}$ such that

the sorted list of priority labels of its immediate successors is lexicographically smaller than the sorted list of priority labels of the immediate successors for all other task $j \in \mathcal{R}$. Break ties arbitrarily. Define $\alpha(j^*) = i$.

3. When all tasks have been labeled, construct a list of tasks $L = (T_n, T_{n-1}, \cdots, T_1)$ such that $\alpha(T_j) = j$ for all j, $1 \leq j \leq n$.

List scheduling (taking distance constraints into account) is applied to list L to generate a Coffman-Graham (CG) schedule.

The CG priority rule does not consider distance constraints when it generates the priority labels. We do not modify the CG priority rule to consider distance constraints. Distance constraints are implicitly considered during list scheduling since only ready tasks are scheduled at each time step.

Approximation Bound for the CG Algorithm

We consider a schedule computed by the CG algorithm and assume by convention the tasks are executed on $P_1, P_2, \ldots$ in decreasing order of CG priority during a given time slot. We note $\text{Next}(T)$ the task executed on P_1 during the time slot $t(T) + 1$, if such a task exists.

A series of *critical tasks* is defined by a right to left scanning of the schedule, i.e., starting with the last scheduled time slot and moving towards the first scheduled time slot. An idle processor during a time slot is considered as an empty task with priority 0. Assume that the critical task U_i, $i \geq 0$, is defined. The critical task U_{i+1} is determined by choosing among candidate critical tasks found as follows.

1. If there exists at least one time slot lower than $t(U_i)$ during which the task executed on P_2 has a priority lower than $l(U_i)$, then the task executed by P_1 during the rightmost of these time slots is a candidate critical task. If no task is executed on P_1, then the leftmost task executed on P_1 before this empty time slot is a candidate critical task.

2. If there exists at least one time slot lower than $t(U_i)$ during which the task T' executed on P_1 has a priority lower than the task executed on P_1 during the

next time slot, then the task T' executed during the rightmost of these time slots is a candidate critical task. Note that T' may correspond to an idle time slot.

3. If $l(U_j) < l(Next(U_j))$ for some $j \in \{0, \ldots, i\}$, there does not exist a $U_k, k \geq j$ such that $l(U_k) \geq l(Next(U_j))$, and there exists at least one time slot lower than $t(U_i)$ during which the task T' executed on P_1 has a priority greater than or equal to the priority of $Next(U_j)$, then the task T' executed during the rightmost of these time slots is a candidate critical task.

The critical task U_{i+1} is defined to be the candidate critical task executed during the rightmost of these time slots. The first critical task U_0 is the task executed by P_1 during the rightmost slot of S_{cg}.

A *block* X_i is defined such that U_{i+1} precedes each task of X_i:

$$X_i = \{T | l(T) \geq l(U_i) \text{ and } t(T) > t(U_{i+1})\}.$$

If U_{i+1} can not be defined, then X_i is defined as:

$$X_i = \{T | l(T) \geq l(U_i)\}.$$

Note that a block spans a contiguous set of time slots, and a block may contain time slots during which no task is scheduled. In particular, a contiguous set of idle time slots is contained within a single block, and the block does not contain any tasks.

Lemma 8.4.4 *At least two tasks of a block X_i are executed during any time slot in which at least one task of block X_i is executed except the last time slot $t(U_i)$.*

Proof Follows from the definition of a block and the way in which critical tasks are determined. ■

Lemma 8.4.5 *In an CG schedule, if T is executed on P_1 and $t(T) \leq t(T')$ for $T, T' \in X_i$ for some block X_i, then $l(T) \geq l(T')$.*

Proof The list L used to construct the CG schedule is in order of decreasing priority values. By convention, processor P_1 is assigned before any of the other processors; therefore, P_1 is always assigned the ready task with the highest priority value. The lemma follows from the definition of critical tasks and blocks. ■

Notice that Lemma 8.4.5 is not necessarily true for the entire schedule when distance constraints are present. This is because distance constraints can cause empty time slots which are then filled in by lower distance level (and lower priority) tasks.

A *fill-in block* is defined to be a block X_j such that U_j is created by condition (2) in the critical task determination or U_{j+1} is created by condition (3) in the critical task determination. Note that fill-in blocks can occur contiguously.

A series of *critical blocks* is defined by a left-to-right scanning of the schedule. Assume that the critical block $Y_j, j \geq 0$, is defined. The set of tasks $W_j(X_i)$ is defined for each block X_i as follows:

$$W_j(X_i) = \{T | T \in Y_j \text{ or } T \text{ is preceded by a task of } Y_j \text{ and } l(T) \geq l(U_{i+1})\}.$$

If there exists at least one block X_i to the right of Y_j such that $W_j(X_i) \prec X_i$ ($W_j(X_i) \prec X_i$ when all of the tasks in X_i are preceded by some task in $W_j(X_i)$), then Y_{j+1} is defined to be the leftmost of these blocks. The *segment* W_j is defined as $W_j(Y_{j+1})$. If there is no block to the right of Y_j such that $W_j(X_i) \prec X_i$, then W_j is defined as follows:

$$W_j = \{T | T \in Y_j \text{ or } T \text{ is preceded by a task of } Y_j \text{ and } l(T) \geq l(U_0).\}.$$

Note that if a segment contains one or more fill-in blocks, the fill-in block(s) occur at the end of the segment. Furthermore, no task of a fill-in block is contained in a segment due to how the tasks are assigned priorities and the definition of a segment.

The first critical block Y_0 is the left most block.

A task executed during a time slot of $w(W_j)$ that does not belong to W_j is called an *extra task,* and a time slot of $w(W_j)$ during which an extra task is executed (or no task is executed) is called a *partial slot.* The p partial slots of $w(W_j)$ are noted

$t_1, \ldots, t_p$ from left to right. The task executed by P_1 during t_i, if it exists, is noted as T_i.

Lemma 8.4.6 *In an MCG schedule, if T is executed on P_1 and $t(T) \leq t(T')$ for $T, T' \in W_j$ for some segment W_j, then $l(T) \geq l(T')$.*

Proof The lemma follows from Lemma 8.4.5 and the definition of segments. ■

Lemma 8.4.7 *A segment W_j is the disjoint union of k consecutive non-fill-in blocks, $k \geq 0$, followed by f consecutive fill-in blocks, $f \geq 0$, with $k + f \geq 1$, and of a set of at least $k - 1$ additional tasks. Each additional task is preceded by a task of W_j.*

Proof By the definition of segments, a segment contains consecutive blocks that are disjoint by the definition of blocks. Fill-in blocks are only contained at the end of a segment, and no task in a fill-in block is part of the segment. Thus, a segment consists of the disjoint union of a (possibly empty) set of non-fill-in blocks followed by the disjoint union of a (possibly empty) set of fill-in blocks with tasks from the time slots corresponding to the non-fill-in blocks being the only tasks contained in the segment. A segment containing zero non-fill-in blocks does not contain any tasks. We prove that a segment containing $k \geq 1$ non-fill-in blocks also contains at least $k - 1$ additional tasks.

Consider segment W_j consisting of k non-fill-in blocks, and let X_i be the second non-fill-in block that is contained in W_j. By assumption on the number of non-fill-in blocks in W_j, $W_j(X_i)$ does not precede all tasks in X_i (a non-fill-in block). Thus, there exists a task T in $W_j(X_i)$ and a task T' in X_i such that T does not precede T'. Choose T such that it has no successors in $W_j(X_i)$ and T' such that it has no predecessors in X_i. Let I be the set of tasks in X_i that have no predecessors in X_i. Clearly $T' \in I$.

Consider U_{i+1}, the last task in X_{i+1}. From the definition of segments $U_{i+1} \in W_j(X_i)$, and from the definition of blocks U_{i+1} precedes all tasks in X_i.

From the definition of segments all tasks in $W_j(X_i)$ have higher priorities than U_{i+1}. Thus, task T has a higher priority than U_{i+1}. Since T has no successors in $W_j(X_i)$, T must either precede all tasks in I or must precede an extra task E for $l(T) \geq l(U_{i+1})$ to be true. Since T does not precede $T' \in I$, E must exist.

E can not be executed after $t(U_{i+1})$. If it were, it would be part of X_i. Thus, E neither belongs to $W_j(X_i)$ nor to X_i. Furthermore, $l(E) \geq l(U_i)$ for $l(T) \geq l(U_{i+1})$. Therefore, the disjoint union of $W_J(X_{i-1})$, X_i, and $\{E\}$ is included in $W_j(X_{i-1})$ if X_{i-1} exists or in W_j if X_{i-1} does not exist.

Repeating this argument for each of the remaining $k-2$ non-fill-in blocks yields $k-1$ extra tasks, each preceded by a task of W_j. The lemma follows. ■

We now show that all tasks in one segment precede all tasks in the next segment. This is Lemma 2.3 in Lam and Sethi. The proof is repeated here for completeness.

Lemma 8.4.8 *For all tasks T_j in W_j and T_{j+1} in W_{j+1}, T_j must be completed before T_{j+1} can start.*

Proof Let X_i be the leftmost task in W_{j+1}. Then from the definition of segments, all tasks T_j in W_j precede all tasks T_{j+1} in X_i. If X_i is the only block in W_{j+1}, then the proof is complete.

Assume that W_{j+1} contains blocks $X_i, X_{i-1}, \cdots, X_{i-k}$, for $k \geq 1$, as well as some extra tasks. From the definition of blocks it follows that for all j, U_j precedes all tasks in X_{j-1}. By transitivity for all T_j in W_j, T_j precedes all tasks in $X_i \cup \cdots \cup X_{i-k}$.

The first extra task E added to W_{j+1} is preceded by some task in X_i. Any subsequent extra task added to W_{j+1} is either preceded by a task in some block in $Wj+1$ or by an extra task already in W_{j+1}. In either case, by transitivity, the extra task is precede by an element of X_i.

The lemma follows. ■

Lemma 8.4.9 *Let W_j be a segment and $t_1, \ldots, t_{p'}$ be the p' partial slots of $t(W_j)$ that occur in the k non-fill-in blocks of W_j. Every task of W_j with a priority greater than or equal to $l(T_i)$ precedes a chain of at least $p' - i$ tasks of W_j.*

Proof Consider the partial slot t_i of W_j that is not the rightmost, if such a slot exists. If the task T_i is a critical task, then it is clear that $T_i \prec Next(T_i)$. If T_i is not a critical task, then because of Lemma 8.4.6 and the block definition, every extra task E executed during t_i is such that $l(E) < l(Next(T_i))$. Thus, T_i has an immediate successor of priority greater than or equal to $l(Next(T_i))$.

Therefore, if T_a is a task of W_j such that $l(T_a) \geq l(T_i)$, then T_a has an immediate successor T_{a+1} such that $l(T_{a+1}) \geq l(Next(T_i))$. By Lemma 8.4.6, $l(T_{a+1}) \geq l(T_{i+1}) \geq l(U^j)$, where U^j is the rightmost task of W_j. Thus, T_{a+1} belongs to W_j. Repeating this argument for t_{i+1} through $t_{p'}$ it is clear that T_a precedes a chain of at least $p' - i$ tasks of W_j. ■

Lemma 8.4.10 *Let W_j be a segment with p partial slots. If W_j starts with a full column, i.e., no idle processors, then an optimal schedule for the tasks in W_j plus the idle partial slots is $w_{opt}(W_j) \geq p + 1$.*

Proof By Lemma 8.4.9 each task T such that $l(T) > l(T_1)$ precedes a chain of at least $p' - 1$ tasks in W_j. Assume that the first time slot of $w(W_j)$ is not a partial slot. If a task T of W_j is such that $t(T) < t(T_1)$ and $l(T) < l(T_1)$, then at least one task of W_j precedes T_1 since the MCG-algorithm computes a list schedule. Therefore, there exists a chain of at least $p' + 1$ tasks, and the optimum schedule must be at least $p' + 1$ time units. If each task T of W_j executed before T_1 is such that $l(T) \geq l(T_1)$, then there exists at least $m + 1$ tasks of W_j preceding a chain of at least $p' - 1$ tasks of W_j and the inequality still holds.

If the set of partial slots includes idle time slots, i.e., time slots during which no task of W_j is executed, then these $p'' = p - p'$ idle partial

slots occur as the last p'' slots of $w(W_j)$. Idle partial slots occur as a result of the non-zero distance constraints. Since the chain of at least $p'+1$ tasks is scheduled before the idle time slots, the last task in the chain must have a distance constraint greater than or equal to p'' to its immediate successors. If this were not the case, then the immediate successors would be scheduled within the fill-in blocks which is not possible due to the definition of critical tasks and the labeling and scheduling techniques used by the CG-algorithm. The optimum schedule for W_j must account for these distance constraints. Therefore, $w_{opt}(W_j) \geq p' + p'' + 1 = p + 1$. ■

Lemma 8.4.11 *Let a schedule of $G = (V, E)$ be computed on m machines by the MCG algorithm and W_j be a segment with respect to this schedule. If $md \geq 1$ or $d = 0$ and $m \geq 2$, then*

$$w(W_j) \leq (2 - 2/(md + m))w_{opt}(W_j) - (m - 2 - F(m))/m$$

where $F(m)$ is equal to 1 if m is odd and greater than 2 and is equal to 0 otherwise.

Proof We consider a segment W_j and we bound the worst makespan $w(W_j)$ the MCG algorithm can compute. Let k be the number of non-fill-in blocks included in W_j, X be the union of the non-fill-in blocks, Z be the union of the fill-in blocks, A be the set $W_j - X$ of additional tasks, ${}^jU$ and Y_j be the leftmost critical task and the leftmost critical block of W_j respectively, and $t_1, \ldots, t_p$ be the p partial slots of $w(W_j)$. We note F, the set of the tasks of X executed during t_1.

We bound Idle(W_j) with respect to p and F. By construction, $w(W_j)$ is equal to $w(X) + w(Z)$, and Idle(W_j) is equal to $mw(Z) + \text{Idle}(X) - |A|$.

The set $w(W_j)$ is divided into four disjoint subsets by checking the types of each time slot:

- $L_D = \{L \in t(W_j) | L$ is a partial slot during which no task of X is executed $\}$,

- $L_1 = \{L \in t(W_j) | L$ is a partial slot during which only one task of X is executed $\}$,
- $L_{1,m} = \{L \in t(W_j) | L$ is a partial slot during which at least two tasks of X are executed $\}$,
- $L_m = \{L \in t(W_j) | L$ is not a partial slot $\}$.

The use of distance constraints requires that

$$|L_D| \leq l_{\max}(W_j)d$$

where $l_{\max}(W_j)$ is the maximum number of distance levels between W_j and W_{j+1} and d is the maximum distance constraint.

Lemma 8.4.4 requires that

$$|L_1| \leq k$$

where k is the number of non-fill-in blocks included in W_j, and the partial slot definition requires that

$$|L_D| + |L_1| + |L_{1,m}| = p$$

where p is the number of partial slots of $w(W_j)$.

The above leads to

$$Idle(W_j) \leq m|L_D| + (m-1)|L_1| + (m-2)|L_{1,m}| - |A|.$$

Using the above inequalities and relationships and substitute $|A|$ by $k-1$, according to Lemma 8.4.7 we obtain:

$$Idle(W_j) \leq (p+1)(m-2) - (m-3) + 2l_{\max}(W_j)d.$$

Assume that the first slot $w(W_j)$ is not partial. By Lemma 8.4.10

$$p + 1 \leq w_{opt}(W_j).$$

If the first slot of $w(W_j)$ is partial, we can derive the following inequality for Idle(W_j):

$$Idle(W_j) \leq Idle(F) + p(m-2) - (m-3) + 2l_{\max}(W_j)d.$$

Lemma 8.4.9 requires that each task T such that $l(T) \geq l(T_1)$ precedes a chain of at least $p' - 1$ tasks of W_j with the last task of the chain containing a distance constraint of at least $p'' = p - p'$. If the first time slot of $w(W_j)$ is partial, there exists a chain of at least p' tasks of W_j with the last task of the chain containing a distance constraint of at least $p'' = p - p'$; therefore,

$$p \leq w_{opt}(W_j).$$

Assume that ${}^jU$ is not executed during t_1. Then, each task of $Y_j - F$ is preceded by at least one task of F. Furthermore, every task of A is preceded by a task of Y_j and ${}^jU$ precedes all tasks of $X - Y_j$. Hence, each task of $W_j - F$ is preceded by a task of F and

$$Idle(F) \leq Idle_{opt}(W_j).$$

If ${}^jU$ is executed during t_1, then $t(W_j)$ is equal to $t({}^jU)$ and the inequality still holds.

Using the expressions $w(W_j) = (|W_j| + Idle(W_j))/m$ and $w_{opt}(W_j) = (|W_j| + Idle_{opt}(W_j))/m$, the above inequalities, and the fact that $w_{opt}(W_j) \geq l_{\max}(W_j)(md + m)/m$ we derive the following inequality:

$$w(W_j) \leq (2 - 2/(md+m))w_{opt}(W_j) - (m-3)/m.$$

When m is even, $(m-3)/m$ cannot be integer, and we can rewrite this equation as

$$w(W_j) \leq (2 - 2/(md+m))w_{opt}(W_j) - (m-2)/m.$$

When m is equal to one (1), the bound determined by Bernstein et. al. [6] is valid:

$$w(W_j) \leq (2 - 2/(d+1))w_{opt}(W_j).$$

■

Theorem 8.4.2 *Let a schedule S_{CG} of the unit execution time task system $(\mathcal{T}, \prec)$ be computed on m machines by the CG algorithm. If $md \geq 1$ or $d = 0$ and $m \geq 2$, then*

$$\omega(S_{CG}) \leq (2 - 2/(md+m))w_{opt}(G) - (m - 2 - F(m))/m,$$

where $F(m)$ is equal to -1 if $m = 1$, 1 if m is odd and greater than 2, and 0 otherwise.

Proof Let there be $r + 1$ segments, $W_0, W_1, \ldots, W_r$, in the CG schedule S. Lemma 8.4.8 shows that an optimal schedule can be no shorter than one that arranges each individual segment optimally. Therefore, letting $w_{opt}(W_j)$ be the optimal schedule length of segment W_j, $0 \leq j \leq r$, we have

$$w_{opt} \geq \sum_{j=0}^{r} w_{opt}(W_j).$$

From Lemma 8.4.11 we have $w(W_j) \leq (2 - 2/(md+m))w_{opt}(W_j) - (m - 2 - F(m))/m$, where $w(W_j)$ is the length of segment W_j in the CG-schedule. Thus,

$$w = \sum_{j=0}^{r} w(W_j) \leq \sum_{j=0}^{r}(2 - 2/(md+m))w_{opt}(W_j) - (m - 2 - F(m))/m,$$

and the theorem is proven. ■

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

J. R. R. Tolkien, *The Lord of the Rings*

Chapter 9

Conclusions

Then our mother came in
And she said to us two,
"Did you have any fun?
Tell me. What did you do?"

And Sally and I did not know
What to say.
Should we tell her
The things that went on there that day?

Should we tell her about it?
Now, what SHOULD we do?
Well ...
What would YOU do
If your mother asked YOU?

Dr. Seuss, *The Cat in the Hat*

The increasing functional complexity of embedded systems is forcing designers to abandon the ad hoc approach to partitioning system functionality. Instead, rigorous approaches that are amenable to automation must be employed. These approaches allow the designer to explore the solution space in a methodical manner. As a result, automated partitioning approaches can provide indispensable feedback on the feasibility of a design and the functional partitions that are most likely to yield optimal implementations.

In this dissertation, I have presented a new approach to solving the hardware-

software partitioning problem in embedded system design and further defined the complexity boundary of this approach. The identification of tasks from the system specification and the scheduling of these tasks form the foundation of my approach. Using a well-studied, effective, and efficient constructive algorithm as the basis to solving the scheduling problem, optimal solutions were obtained for several examples from the literature. Furthermore, applying my approach to the wristwatch example illustrated its effectiveness in exploring the solution space of a design. My scheduling-based approach to solving the hardware-software partitioning problem is a simple, fast, and effective approach to performing automated analysis of a system design and quickly exploring the solution space.

Tasks modeling a system specification are created by choosing an appropriate set of task regions, each clustering a disjoint set of system functionality. I presented a new characterization for task regions based upon single-entry single-exit regions of the control flow graph. The use of single-entry single-exit regions as the foundation for the identification of task regions provides a systematic and rigorous method of clustering functionality into tasks. This characterization is stronger than previous attempts at generating tasks from a system specification in that it considers the dynamic execution properties of the tasks. My characterization of static task regions guarantees that the entry and exit edges execute the same number of times, with control never passing through one of the edges twice without first passing through the other. In addition, static task regions contain all control paths between `switch` and `merge` nodes in the control flow graph. These properties allow static task regions to be used within traditional scheduling problem formulations that do not permit control-flow dependent executions.

In order to simplify the identification of tasks from a system specification, I defined the notion of canonical static task regions and proved that they are either node disjoint or nested. By modeling only canonical static task regions as tasks, the number of task regions that need to be considered in determining a set of tasks that completely model the entire system is reduced. The use of canonical static task regions allows a process to be modeled by a single chain of tasks. Thus, a system specification is

modeled as a set of task chains.

My task region-based approach to determining clusters of functionality is general enough to be used with any system specification language. It may also be used to determine tasks where the target system allows dynamic scheduling of the system functionality. For this target architecture, every task region and every `switch` and `merge` node in the control-flow graph may correspond to a task (not just static task regions). Furthermore, the task region properties force well-defined interfaces between tasks, simplifying the required communication between tasks.

My pure scheduling problem formulation simultaneously solves both the allocation and the scheduling subproblems of the hardware-software partitioning problem. This formulation is able to accurately model the partitioning problem at all levels of abstraction, and a solution to the scheduling problem yields a deterministic schedule of the tasks that the designer can use to analyze the system design. A quick glance at the complexity boundary surrounding my scheduling problem formulation shows that problem formulations that are easier to solve can be obtained by slightly changing the formulation or even the system specification. These changes can allow more efficient or even optimal algorithms to be used to determine a solution.

As we saw in Chapter 6, my simple problem formulation can be used to solve more complex hardware-software partitioning problems. A simple algorithm based on my scheduling problem formulation was able to effectively determine a set of processing elements upon which to execute the system functionality.

The main disadvantage of the scheduling problem formulation approach is that hard system constraints such as area and power cannot be modeled. These constraints must be checked after a solution to the scheduling problem has been determined. Therefore, this approach may perform poorly when the solution to the partitioning problem is dictated primarily by area and/or power constraints instead of timing constraints as the formulation assumes.

Appendix

Appendix A

Digital Wristwatch Example

This appendix contains the Scenic specification for the digital wristwatch example used in this dissertation.

Wristwatch Example Main Scenic File

```
1:   int scenic( int ac, char* av[]) {              // Main Method for Declaring a Wristwatch
2:     // wristwatch button inputs
3:     sc_signal<std_ulogic> UL;                    // Upper Left Button
4:     sc_signal<std_ulogic> LL;                    // Lower Left Button
5:     sc_signal<std_ulogic> UR;                    // Upper Right Button
6:     sc_signal<std_ulogic> LR;                    // Lower Right Button
7:
8:     // internal signals
9:     sc_signal<std_ulogic> watchTime;             // Current Time
10:    sc_signal<std_ulogic> newWatchTime;          // Time After Increment
11:    sc_signal<std_ulogic> alarmTime;             // Alarm Time
12:    sc_signal<std_ulogic> stopTime;              // Stopwatch Time
13:    sc_signal<std_ulogic> newStopTime;           // Stopwatch Time After Increment
14:    sc_signal<std_ulogic> toggleBeep;            // Toggle Hourly Chime On/Off
15:    sc_signal<std_ulogic> watchBeep;             // Chime if on Hour and Beep On
16:    sc_signal<std_ulogic> alarmBeep;             // Chime if Alarm Time and Alarm On
17:    sc_signal<std_ulogic> toggleAlarm;           // Toggle Alarm On/Off
18:    sc_signal<std_ulogic> setWatchPosition;      // Increment Position Value in Set Mode
19:    sc_signal<std_ulogic> nextWatchPosition;     // Move to Next Position in Set Mode
20:    sc_signal<std_ulogic> startStop;             // Start/Stop Stopwatch
21:    sc_signal<std_ulogic> stopReset;             // Reset Stopwatch to Zero
22:    sc_signal<std_ulogic> stopLap;               // Display Stopwatch Lap Value
23:    sc_signal<std_ulogic> displayMode;           // Display Mode:  watch, alarm, stopwatch
24:
25:    // wristwatch functional outputs
26:    sc_signal<std_ulogic> beepStatus;
27:    sc_signal<std_ulogic> mainDisplay;
28:
29:    // declare the clock
30:    sc_clock clk(''CLOCK'', 100.0, 0.5, 0.0);
31:
32:    // declare the processes
33:    Button Btn(''Button'', clk.pos(),
34:      // inputs
35:      UL, LL, UR, LR,
36:      // outputs
37:      toggleBeep, toggleAlarm, displayMode, nextWatchPosition, setWatchPosition, startStop, stopReset, stopLap);
38:    SetWatch SWatch(''SetWatch'', clk.pos(),
39:      // inputs
40:      displayMode, nextWatchPosition, setWatchPosition, newWatchTime,
41:      // outputs
42:      watchTime);
43:    BasicWatch BWatch(''BasicWatch'', clk.pos(),
44:      // inputs
```

```
45:            toggleBeep, watchTime,
46:            // outputs
47:            watchBeep, newWatchTime);
48:         SetAlarm SetA(''SetAlarm'', clk.pos(),
49:            // inputs
50:            displayMode, nextWatchPosition, setWatchPosition,
51:            // outputs
52:            alarmTime);
53:         Alarm A(''Alarm'', clk.pos(),
54:            // inputs
55:            toggleAlarm, alarmTime, newWatchTime,
56:            // outputs
57:            alarmBeep);
58:         BasicStopWatch BStop(''BasicStopWatch'', clk.pos(),
59:            // inputs
60:            displayMode, startStop, stopReset,
61:            // outputs
62:            newStopTime);
63:         LapFilter Lap(''LapFilter'', clk.pos(),
64:            // inputs
65:            newStopTime, stopLap,
66:            // outputs
67:            stopTime);
68:         Beep B(''Beep'', clk.pos(),
69:            // inputs
70:            watchBeep, alarmBeep, newWatchTime
71:            // outputs
72:            beepStatus);
73:         Display D(''Display'', clk.pos(),
74:            // inputs
75:            displayMode, newWatchTime, stopTime, alarmTime,
76:            // outputs
77:            mainDisplay);
78:      }
```

Button Scenic Process Definition

```
1:       struct Button:  public sc_sync {                      // Definition of Button Process
2:          // input ports
3:          const sc_signal<std_ulogic>& UL;                   // Upper Left Button
4:          const sc_signal<std_ulogic>& LL;                   // Lower Left Button
5:          const sc_signal<std_ulogic>& UR;                   // Upper Right Button
6:          const sc_signal<std_ulogic>& LR;                   // Lower Right Button
7:          // output ports
8:          sc_signal<std_ulogic>& toggleBeep;                 // Toggle Hourly Beep
9:          sc_signal<std_ulogic>& toggleAlarm;                // Toggle Alarm ON/OFF
10:         sc_signal<std_ulogic>& displayMode;                // What to Display
11:         sc_signal<std_ulogic>& nextWatchPosition;          // Next Position in Set Watch/Alarm
12:         sc_signal<std_ulogic>& setWatchPosition;           // Set Position in Set Watch/Alarm
13:         sc_signal<std_ulogic>& startStop;                  // START/STOP Stopwatch
14:         sc_signal<std_ulogic>& stopReset;                  // Reset Stopwatch Time
15:         sc_signal<std_ulogic>& stopLap;                    // Stopwatch Lap Display
16:         // internal variables to this process
17:         int currentMode;                                   // Current Operation Mode
18:
19:         // The constructor
20:         Button( sc_clock_edge& EDGE,                       // Constructor and its Parameters
21:            sc_signal<std_ulogic>& TOGGLE_BEEP,
22:            sc_signal<std_ulogic>& TOGGLE_ALARM,
23:            sc_signal<std_ulogic>& DISPLAY_MODE,
24:            sc_signal<std_ulogic>& NEXT_WATCH_POSITION,
25:            sc_signal<std_ulogic>& SET_WATCH_POSITION,
26:            sc_signal<std_ulogic>& START_STOP,
27:            sc_signal<std_ulogic>& STOP_RESET,
28:            sc_signal<std_ulogic>& STOP_LAP)
29:         :  sc_sync(EDGE),
30:            toggleBeep(TOGGLE_BEEP),
31:            toggleAlarm(TOGGLE_ALARM),
32:            displayMode(DISPLAY_MODE),
33:            nextWatchPosition(NEXT_WATCH_POSITION),
34:            setWatchPosition(SET_WATCH_POSITION),
35:            startStop(START_STOP),
36:            stopReset(STOP_RESET),
37:            stopLap(STOP_LAP)
38:         {
39:            toggleBeep.write( 0 );                          // Don't Change Default Beep ON/OFF
40:            toggleAlarm.write( 0 );                         // Don't Change Default Alarm ON/OFF
41:            displayMode.write( 0 );                         // Display Time
```

```
42:          nextWatchPosition.write( 0 );                 // Don't Change Watch Position
43:          setWatchPosition.write( 0 );                  // Don't Set Watch Position
44:          startStop.write( 0 );                         // Don't START/STOP the Stopwatch
45:          stopReset.write( 0 );                         // Don't Reset Stopwatch
46:          stopLap.write( 0 );                           // Don't Display Stopwatch Lap
47:          currentMode = 0;                              // Display the Time
48:       }
49:       void entry();                                    // Process Functionality Contained Here
50:    };
51:
52:    void Button::entry()                                // Definition of Process Functionality
53:    {
54:       ul = UL.read();                                  // Read the UL Button and Store its Value
55:       ll = LL.read();                                  // Read the LL Button and Store its Value
56:       ur = UR.read();                                  // Read the UR Button and Store its Value
57:       lr = LR.read();                                  // Read the LR Button and Store its Value
58:       nextMode = 0;                                    // Next Mode of Operation
59:
60:       while( true ) {                                  // This Process Runs Forever
61:          if( currentMode == 0 ) {                      // Watch Mode
62:             nextMode = 0;
63:             if( ll == 1 ) {
64:                nextMode = 1;                           // Next Mode is Stopwatch
65:             }
66:             if( ul == 1 ) {
67:                nextMode =3;                            // Next Mode is Set Watch
68:             }
69:          }
70:
71:          if( currentMode == 1 ) {                      // Stopwatch Mode
72:             nextMode = 1;
73:             if( lr == 1 ) {
74:                startStop.write(1);
75:             }
76:             if( ur == 1 ) {
77:                stopLap.write(1);
78:             }
79:             if( ul == 1 ) {
80:                stopReset.write(1);
81:             }
82:             if( ll == 1 ) {
83:                nextMode = 2;                           // Next Mode is Alarm
84:             }
85:          }
86:
87:          if( currentMode == 2 ) {                      // Alarm Mode
88:             nextMode = 2;
89:             if( ur == 1 ) {
90:                toggleAlarm.write(1);
91:             }
92:             if( ul == 1 ) {
93:                nextMode = 4;                           // Next Mode is Set Alarm
94:             }
95:             if( ll == 1 ) {
96:                nextMode = 0;                           // Next Mode is Watch
97:             }
100:         }
101:
102:         if( currentMode == 3 ) {                      // Set Watch Mode
103:            nextMode = 3;
104:            if( ul == 1 ) {
105:               nextMode = 0;                           // Next Mode is Watch
106:            }
107:            if( lr == 1 ) {
108:               setWatchPosition.write(1);
109:            }
110:            if( ll == 1 ) {
111:               nextWatchPosition.write(1);
112:            }
113:         }
114:
115:         if( currentMode == 4 ) {                      // Set Alarm Mode
116:            nextMode = 4;
117:            if( ul == 1 ) {
118:               nextMode = 2;                           // Next Mode is Alarm
119:            }
120:            if( lr == 1 ) {
121:               setWatchPosition.write(1);
122:            }
123:            if( ll == 1 ) {
124:               nextWatchPosition.write(1);
125:            }
126:         }
127:
128:         currentMode = nextMode;
129:         wait();
130:      }
131:   }
```

SetWatch Scenic Process Definition

```
1:    struct SetWatch:  public sc_sync {                     // Definition of SetWatch Process
2:       // input ports
3:       const sc_signal<std_ulogic>& displayMode;           // Check if in Set Watch Mode
4:       const sc_signal<std_ulogic>& nextWatchPosition;     // Change Position Being Set
5:       const sc_signal<std_ulogic>& setWatchPosition;      // Increment Current Position Value
6:       const sc_signal<std_ulogic>& newWatchTime;          // Current Watch Time
7:       // output ports
8:       sc_signal<std_ulogic>& watchTime;                   // Current Watch Time
9:       // internal variables to this process
10:      int currentPosition;                                // Position Being Set
11:      int time;                                           // Temporary Time Var
12:
13:      // The constructor
14:      SetWatch( sc_clock_edge& EDGE,                      // Constructor and its Parameters
15:         sc_signal<std_ulogic>& DISPLAY_MODE,
16:         sc_signal<std_ulogic>& NEXT_WATCH_POSITION,
17:         sc_signal<std_ulogic>& SET_WATCH_POSITION,
18:         sc_signal<std_ulogic>& NEW_WATCH_TIME,
19:         sc_signal<std_ulogic>& WATCH_TIME)
20:      :  sc_sync(EDGE),
21:         displayMode(DISPLAY_MODE),
22:         nextWatchPosition(NEXT_WATCH_POSITION),
23:         setWatchPosition(SET_WATCH_POSITION),
24:         newWatchTime(NEW_WATCH_TIME),
25:         watchTime(WATCH_TIME)
26:      {
27:         currentPosition = 0;                             // Start with the Hour
28:         time = 0;                                        // No Assumption of Time
29:      }
30:      void entry();                                       // Process Functionality Contained Here
31:   };
32:
33:   void SetWatch::entry()                                 // Definition of Process Functionality
34:   {
35:      int inSetWatchMode;
36:
37:      while( true ) {                                     // This Process Runs Forever
38:         if( displayMode.read() == 3 ) {
39:            inSetWatchMode = 1;
40:            time = newWatchTime.read();
41:         } else {
42:            inSetWatchMode = 0;
43:            time = newWatchTime.read();
44:         }
45:         if( (setWatchPosition.read() == 1) && (inSetWatchMode == 1) ) {
46:            time = SetTime( newWatchTime.read(), currentPosition );
47:         }
48:         if( (nextWatchPosition.read() == 1) && (inSetWatchMode == 1) ) {
49:            currentPosition = NextPosition( currentPosition );
50:         }
51:         watchTime.write( time );
52:         wait();
53:      }
54:   }
```

BasicWatch Scenic Process Definition

```
1:    struct BasicWatch:  public sc_sync {                   // Definition of BasicWatch Process
2:       // input ports
3:       const sc_signal<std_ulogic>& toggleBeep;            // Toggle Hourly Beep
4:       const sc_signal<std_ulogic>& watchTime;             // Current Watch Time
5:       // output ports
6:       sc_signal<std_ulogic>& watchBeep;                   // Beep if Top of Hour and Beep Set
7:       sc_signal<std_ulogic>& newWatchTime;                // Current Watch Time Plus 1 Clock Cycle
8:       // internal variables to this process
9:       int beepStatus;                                     // To Beep or Not to Beep
10:
11:      // The constructor
12:      BasicWatch( sc_clock_edge& EDGE,                    // Constructor and its Parameters
13:         sc_signal<std_ulogic>& TOGGLE_BEEP,
14:         sc_signal<std_ulogic>& WATCH_BEEP,
```

```
15:            sc_signal<std_ulogic>& WATCH_TIME,
16:            sc_signal<std_ulogic>& NEW_WATCH_TIME)
17:         :  sc_sync(EDGE),
18:            toggleBeep(TOGGLE_BEEP),
19:            watchBeep(WATCH_BEEP),
20:            watchTime(WATCH_TIME),
21:            newWatchTime(NEW_WATCH_TIME)
22:         {
23:            newWatchTime.write( 0 );                    // Start the Watch at 0 Hour
24:            beepStatus = 0;                             // Don't Beep on the Hour
25:         }
26:         void entry();                                  // Process Functionality Contained Here
27:      };
28:
29:      void BasicWatch::entry()                          // Definition of Process Functionality
30:      {
31:         int numBeeps;
32:         int time;
33:
34:         while( true ) {                                // This Process Runs Forever
35:            if (toggleBeep.read == '1') {               // Toggle Hourly Beep Status
36:               if (beepStatus == 0)
37:                  beepStatus = 1;
38:               else
39:                  beepStatus = 0;
40:            }
41:            time = IncrementWatchTime( watchTime.read );   // Update Watch Time
42:            newWatchTime.write( time );
43:            numBeeps = Beep( time, beepstatus );        // If on the Hour, then Beep
44:            watchBeep.write( numBeeps );
45:            wait();                                     // Pause Until Next Clock Edge
46:         }
47:      }
```

SetAlarm Scenic Process Definition

```
1:       struct SetAlarm:  public sc_sync {                // Definition of SetAlarm Process
2:          // input ports
3:          const sc_signal<std_ulogic>& displayMode;      // Check if in Set Alarm Mode
4:          const sc_signal<std_ulogic>& nextWatchPosition;   // Change Position Being Set
5:          const sc_signal<std_ulogic>& setWatchPosition;    // Increment Current Position Value
6:          // output ports
7:          sc_signal<std_ulogic>& alarmTime;              // Current Alarm Time
8:          // internal variables to this process
9:          int currentPosition;                           // Position Being Set
10:         int time;                                      // Alarm Time Var
11:
12:         // The constructor
13:         SetAlarm( sc_clock_edge& EDGE,                 // Constructor and its Parameters
14:            sc_signal<std_ulogic>& DISPLAY_MODE,
15:            sc_signal<std_ulogic>& NEXT_WATCH_POSITION,
16:            sc_signal<std_ulogic>& SET_WATCH_POSITION,
17:            sc_signal<std_ulogic>& ALARM_TIME)
18:         :  sc_sync(EDGE),
19:            displayMode(DISPLAY_MODE),
20:            nextWatchPosition(NEXT_WATCH_POSITION),
21:            setWatchPosition(SET_WATCH_POSITION),
22:            alarmTime(ALARM_TIME)
23:         {
24:            currentPosition = 0;                        // Start with the Hour
25:            time = 0;                                   // No Assumption of Alarm Time
26:         }
27:         void entry();                                  // Process Functionality Contained Here
28:      };
29:
30:      void SetAlarm::entry()                            // Definition of Process Functionality
31:      {
32:         int inSetAlarmMode;
33:
34:         while( true ) {                                // This Process Runs Forever
35:            if( displayMode.read() == 4 ) {
36:               inSetAlarmMode = 1;
37:            } else {
38:               inSetAlarmMode = 0;
39:            }
40:            if( (setWatchPosition.read() == 1) && (inSetAlarmMode == 1) ) {
41:               time = SetTime( time, currentPosition );
42:            }
```

```
43:           if( (nextWatchPosition.read() == 1) && (inSetAlarmMode == 1) ) {
44:              currentPosition = NextPosition( currentPosition );
45:           }
46:           alarmTime.write( time );
47:           wait();
48:        }
49:     }
```

Alarm Scenic Process Definition

```
1:      struct Alarm:  public sc_sync {                    // Definition of Alarm Process
2:         // input ports
3:         const sc_signal<std_ulogic>& toggleAlarm;       // Toggle Alarm ON/OFF
4:         const sc_signal<std_ulogic>& alarmTime;         // Alarm Time
5:         const sc_signal<std_ulogic>& newWatchTime;      // Current Watch Time
6:         // output ports
7:         sc_signal<std_ulogic>& alarmBeep;               // Beep if Alarm Time
8:         // internal variables to this process
9:         int alarmOnOff;                                 // Alarm Status
10:
11:        // The constructor
12:        Alarm( sc_clock_edge& EDGE,                     // Constructor and its Parameters
13:           sc_signal<std_ulogic>& TOGGLE_ALARM,
14:           sc_signal<std_ulogic>& ALARM_TIME,
15:           sc_signal<std_ulogic>& NEW_WATCH_TIME,
16:           sc_signal<std_ulogic>& ALARM_BEEP)
17:        :  sc_sync(EDGE),
18:           toggleAlarm(TOGGLE_ALARM),
19:           alarmTime(ALARM_TIME),
20:           newWatchTime(NEW_WATCH_TIME),
21:           alarmBeep(ALARM_BEEP)
22:        {
23:           alarmOnOff = 0;                              // Start with Alarm Off
24:        }
25:        void entry();                                   // Process Functionality Contained Here
26:     };
27:
28:     void Alarm::entry()                                // Definition of Process Functionality
29:     {
30:        while( true ) {                                 // This Process Runs Forever
31:           if( toggleAlarm.read() == 1 ) {
32:              if( alarmOnOff == 0 ) {
33:                 alarmOnOff = 1;
34:              } else {
35:                 alarmOnOff = 0;
36:              }
37:           } else
38:           if( alarmOnOff == 1 ) {
39:              if( alarmTime.read() == newWatchTime.read() ) {
40:                 alarmBeep.write( 30 );
41:              }
42:           }
43:           wait();
44:        }
45:     }
```

BasicStopWatch Scenic Process Definition

```
1:      struct BasicStopWatch:  public sc_sync {           // Definition of BasicStopWatch Process
2:         // input ports
3:         const sc_signal<std_ulogic>& displayMode;       // Check if in Stopwatch Mode
4:         const sc_signal<std_ulogic>& startStop;         // START/STOP Stopwatch
5:         const sc_signal<std_ulogic>& stopReset;         // Reset Stopwatch Time
6:         // output ports
7:         sc_signal<std_ulogic>& newStopTime;             // Current Stopwatch Time
```

```
8:          // internal variables to this process
9:          int stopTime;                                   // Stopwatch Time Var
10:         int run;                                        // Running Time
11:
12:         // The constructor
13:         BasicStopWatch( sc_clock_edge& EDGE,            // Constructor and its Parameters
14:            sc_signal<std_ulogic>& DISPLAY_MODE,
15:            sc_signal<std_ulogic>& START_STOP,
16:            sc_signal<std_ulogic>& STOP_RESET,
17:            sc_signal<std_ulogic>& NEW_STOP_TIME)
18:         :  sc_sync(EDGE),
19:            displayMode(DISPLAY_MODE),
20:            startStop(START_STOP),
21:            stopReset(STOP_RESET),
22:            newStopTime(NEW_STOP_TIME)
23:         {
24:            stopTime = 0;                                // Start Stopwatch at Time 0
25:            run = 0;                                     // Not Running
26:         }
27:         void entry();                                   // Process Functionality Contained Here
28:      };
29:
30:      void BasicStopWatch::entry()                       // Definition of Process Functionality
31:      {
32:         while( true ) {                                 // This Process Runs Forever
33:            if( run == 1 ) {
34:               stopTime = IncrementStopTime( stopTime );
35:            }
36:
37:            if( startStop.read() == 1 ) {
38:               if( run == 0 ) {
39:                  run = 1;
40:               } else {
41:                  run = 0;
42:               }
43:            } else
44:            if( stopReset.read() == 1 ) {
45:               run = 0;
46:               stopTime = 0;
47:            }
48:            newStopTime.write( stopTime );
49:            wait();
50:         }
51:      }
```

LapFilter Scenic Process Definition

```
1:       struct LapFilter:  public sc_sync {                // Definition of LapFilter Process
2:          // input ports
3:          const sc_signal<std_ulogic>& newStopTime;       // Stopwatch Time
4:          const sc_signal<std_ulogic>& stopLap;           // Use Lap Filter
5:          // output ports
6:          sc_signal<std_ulogic>& stopTime;                // Stopwatch Time to Display
7:          // internal variables to this process
8:          int lapFilter;                                  // Apply Lap Filter Var
9:          int lapTime;                                    // Lap Filter Time to Display
10:
11:         // The constructor
12:         LapFilter( sc_clock_edge& EDGE,                 // Constructor and its Parameters
13:            sc_signal<std_ulogic>& NEW_STOP_TIME,
14:            sc_signal<std_ulogic>& STOP_LAP,
15:            sc_signal<std_ulogic>& STOP_TIME)
16:         :  sc_sync(EDGE),
17:            newStopTime(NEW_STOP_TIME),
18:            stopLap(STOP_LAP),
19:            stopTime(STOP_TIME)
20:         {
21:            lapFilter = 0;                               // Display Current Stopwatch Time
22:         }
23:         void entry();                                   // Process Functionality Contained Here
24:      };
25:
26:      void LapFilter::entry()                            // Definition of Process Functionality
27:      {
28:         while( true ) {                                 // This Process Runs Forever
29:            if( stopLap.read() == 1 ) {
30:               if( lapFilter == 0 ) {
31:                  lapFilter = 1;
```

```
32:                 lapTime = newStopTime.read();
33:              } else {
34:                 lapFilter = 0;
35:              }
36:           }
37:
38:           If( lapFilter == 1 ) {
39:              stopTime.write( lapTime );
40:           } else {
41:              stopTime.write( newStopTime.read();
42:           }
43:
44:           wait();
45:        }
46:     }
```

Beep Scenic Process Definition

```
1:      struct Beep:  public sc_sync {                        // Definition of Beep Process
2:         // input ports
3:         const sc_signal<std_ulogic>& watchBeep;            // Beep From Watch
4:         const sc_signal<std_ulogic>& alarmBeep;            // Beep From Alarm
5:         const sc_signal<std_ulogic>& newWatchTime;         // Current Watch Time
6:         // output ports
7:         sc_signal<std_ulogic>& beepStatus;                 // 1 = Beep, 0 otherwise
8:         // internal variables to this process
9:         int numBeeps;                                      // Number of Beeps Left
10:
11:        // The constructor
12:        Beep( sc_clock_edge& EDGE,                         // Constructor and its Parameters
13:           sc_signal<std_ulogic>& WATCH_BEEP,
14:           sc_signal<std_ulogic>& ALARM_BEEP,
15:           sc_signal<std_ulogic>& NEW_WATCH_TIME,
16:           sc_signal<std_ulogic>& BEEP_STATUS)
17:        :  sc_sync(EDGE),
18:           watchBeep(WATCH_BEEP),
19:           alarmBeep(ALARM_BEEP),
20:           newWatchTime(NEW_WATCH_TIME),
21:           beepStatus(BEEP_STATUS)
22:        {
23:           numBeeps = 0;                                   // No Beeps to Start
24:        }
25:        void entry();                                      // Process Functionality Contained Here
26:     };
27:
28:     void Beep::entry()                                    // Definition of Process Functionality
29:     {
30:        while( true ) {                                    // This Process Runs Forever
31:           if( (numBeeps > 0) && (IsSecond(newWatchTime.read())) ) {
32:              beepStatus.write( 1 );                       // Beep Only on Second
33:              numBeeps -= 1;
34:           }
35:
36:           wait();
37:        }
38:     }
```

Display Scenic process definition

```
1:      struct Display:  public sc_sync {                     // Definition of Display Process
2:         // input ports
3:         const sc_signal<std_ulogic>& displayMode;          // Display Mode
4:         const sc_signal<std_ulogic>& newWatchTime;         // Current Watch Time
5:         const sc_signal<std_ulogic>& stopTime;             // Stopwatch Time to Display
6:         const sc_signal<std_ulogic>& alarmTime;            // Alarm Time to Display
```

```
7:          // output ports
8:          sc_signal<std_ulogic>& mainDisplay;              // Digital Wristwatch Display
9:          // internal variables to this process
10:
11:         // The constructor
12:         Display( sc_clock_edge& EDGE,                    // Constructor and its Parameters
13:            sc_signal<std_ulogic>& DISPLAY_MODE,
14:            sc_signal<std_ulogic>& NEW_WATCH_TIME,
15:            sc_signal<std_ulogic>& STOP_TIME,
16:            sc_signal<std_ulogic>& ALARM_TIME,
17:            sc_signal<std_ulogic>& MAIN_DISPLAY)
18:         :  sc_sync(EDGE),
19:            displayMode(DISPLAY_MODE),
20:            newWatchTime(NEW_WATCH_TIME),
21:            stopTime(STOP_TIME),
22:            alarmTime(ALARM_TIME),
23:            mainDisplay(MAIN_DISPLAY)
24:         {
25:            mainDisplay.write( watchDisplay() );          // Display Watch
26:         }
27:         void entry();                                    // Process Functionality Contained Here
28:      };
29:
30:      void Display::entry()                               // Definition of Process Functionality
31:      {
32:         int3mode;
34:         while( true ) {                                  // This Process Runs Forever
35:            mode = displayMode.read();
36:            if( (mode == 0) || (mode == 3) ) {
37:               mainDisplay.write( watchDisplay() );
38:            }
39:            if( mode == 1 ) {
40:               mainDisplay.write( stopWatchDisplay() );
41:            }
42:            if( (mode == 2) || (mode == 4) ) {
43:               mainDisplay.write( alarmDisplay() );
44:            }
45:
46:            wait();
47:         }
48:      }
```

Bibliography

[1] Esther M. Arkin and Robin O. Roundy. Weighted-tardiness scheduling on parallel machines with proportional weights. *Operations Research*, 39(1):64–81, January-February 1991.

[2] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies. In *Proceedings of the International Workshop on Hardware/Software Codesign*, pages 161–165, March 1997.

[3] Kenneth R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[4] Egon Balas, Jan Karel Lenstra, and Alkis Vazacopoulos. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science*, 41(1):94–109, 1995.

[5] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[6] David Bernstein, Michael Rodeh, and Izidor Gertner. Approximation algorithms for scheduling arithmetic expressions on pipelined machines. *Journal of Algorithms*, 10:120–139, 1989.

[7] J. Blȧzewicz, K.H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes.* Springer, 1996.

[8] Peter Brucker, Thomas Hilbig, and Johann Hurink. A branch and bound algorithm for a single-machine scheduling problem with positive and negative time-lags. Technical Report Reihe P, Nr. 179, Universitat Osnabruck, Osnabrucker Schriften zur Mathematik, 1997.

[9] Peter Brucker and Sigrid Knust. Complexity results for single-machine problems with positive finish-start time-lags. Technical Report Reihe P, Heft 202, Universitat Osnabruck, Osnabrucker Schriften zur Mathematik, 1998.

[10] John Bruno, John W. Jones, III, and Kimming So. Deterministic scheduling with pipelined processors. *IEEE Transactions on Computers*, C-29(4):308–316, April 1980.

[11] L. Bruno, Jr. E.G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[12] D.C. Carroll. *Heuristic Sequencing of Jobs with Single and Multiple Components.* PhD thesis, Sloan School of Management, Massachusetts Institute of Technology, 1965.

[13] E.G. Coffman, Jr. and R.L. Graham. Optimal sequencing for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[14] Joseph G. D'Ambrosio and Xiaobo (Sharon) Hu. Configuration-level hardware/software partitioning for real-time embedded systems. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 34–41, September 22-24 1994.

[15] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *Proceedings of the 34th Design Automation Conference*, pages 703–708, June 1997.

[16] Giovanni DeMicheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., 1994.

[17] Robert P. Dick and Niraj K. Jha. MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, October 1998.

[18] Jianzhong Du and Joseph Y-T. Leung. Minimizing total tardiness on one machine is np-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.

[19] Jianzhong Du, Joseph Y-T. Leung, and Gilbert H. Young. Scheduling chain-structured tasks to minimize makespan and mean flow time. *Information and Computation*, 92:219–236, 1991.

[20] Jr. E.G. Coffman, editor. *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, 1976.

[21] Petru Eles, Zebo Peng, , Krzysztof Kuchcinski, and Alexa Doboli. Hardware/software partitioning of VHDL system specifications. In *Proceedings of European Design Automation Conference*, pages 434–439, September 1996.

[22] Daniel W. Engels, David R. Karger, Stavros G. Kolliopoulos, Sudipta Sengupta, R.N. Uma, and Joel Wein. Techniques for scheduling with rejection. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *ESA '98, Proceedings of the 6th Annual European Symposium on Algorithms, LNCS*, pages 490–501. Springer, 1998.

[23] Lucian Finta and Zhen Liu. Single machine scheduling subject to precedence delays. *Discrete Applied Mathematics*, 70:247–266, 1996.

[24] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[25] Abhijit Ghosh, Alan Ahlschlager, and Stan Liao. *Using C++ and Scenery for Describing Systems and Hardware.* Synopsys, Inc., 0.7 edition, 1998.

[26] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[27] R.L. Graham. Bounds on the performance of scheduling algorithms. In E.G. Coffman, editor, *Computer and Job-Shop Scheduling Theory*, pages 165–227. John Wiley, New York, 1976.

[28] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[29] Jörg Henkel and Rolf Ernst. A hardware/software partitioner using a dynamically determined granularity. In *Proceedings of the 34th Design Automation Conference*, pages 691–696, June 1997.

[30] Junwei Hou and Wayne Wolf. Process partitioning for distributed embedded systems. In *Proceedings of the Fourth International Workshop on Hardware/Software Codesign*, pages 70–76, March 1996.

[31] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[32] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975.

[33] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, June 1994.

[34] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 42–48, September 22-24 1994.

[35] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[36] Peter Voigt Knudsen and Jan Madsen. PACE: A dynamic programming algorithm for hardware/software partitioning. In *Proceedings of the Fourth International Workshop on Hardware/Software Codesign*, pages 85–92, 1996.

[37] E. Kutanoglu and I. Sabuncuoglu. An analysis of heuristics in a dynamic job shop with weighted tardiness objectives. *International Journal of Production Research*, 37(1):165–187, 1999.

[38] E.L. Lawler and C. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 12, February 1981.

[39] E.L. Lawler and J.M. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16(1), September 1969.

[40] Eugene L. Lawler. Fast approximation algorithms for knapsack problems. In *18th Annual Symposium on Foundations of Computer Science*, pages 206–213, 1977.

[41] Eugene L. Lawler. A 'pseudopolynomial' algorithm for sequencing jobs to minimize total tardiness. In *Annals of Discrete Mathematics 1: Studies in Integer Programming*, pages 331–342. North-Holland Publishing Company, 1977.

[42] Eugene L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[43] Hon F. Li. Scheduling trees in parallel/pipelined processing environments. *IEEE Transactions on Computers*, C-26(11):1101–1112, 1977.

[44] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[45] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. In *Proceedings of IEEE/ACM International Conference on Computer Aided Design*, pages 380–387, November 1995.

[46] S. Miyazaki. Combined scheduling system for reducing job tardiness in a job shop. *International Journal of Production Research*, 19:201–211, 1981.

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

[48] Thomas E. Morton and Ram Mohan V. Rachamadugu. Myopic heuristics for the single machine weighted tardiness problem. Technical Report CMU-RI-TR-83-09, Robotics Institute, Carnegie Mellon University, November 1982.

[49] Thomas E. Morton and P. Ramnath. Guided forward tabu/beam search for scheduling very large dynamic job shops, i. Technical Report 1992-47, Graduate School of Industrial Administration, Carnegie Mellon University, 1992.

[50] Alix Munier, Maurice Queyranne, and Andreas S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In R.E. Bixby, E.A. Boyd, and R.Z. Rios-Mercado, editors, *IPCO VI LNCS 1412*, pages 367–382, Berlin, 1998. Springer-Verlag.

[51] Hyunok Oh and Soonhoi Ha. A hardware-software cosynthesis technique based on heterogeneous multiprocessor scheduling. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, pages 183–187, May 1999.

[52] Kunle A. Olukotun, Rachid Helaihel, Jeremy Levitt, and Ricardo Ramirez. A software-hardware cosynthesis approach to digital system simulation. *IEEE Micro*, 14(4):48 – 58, August 1994.

[53] Krishna V. Palem and Barbara B. Simons. Scheduling time-critical instructions on RISC machines. In *Proceedings of the 17th Annual Symposium on Principles of Programming Languages*, pages 270–280, 1990.

[54] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, 1995.

[55] Shiv Prakash and Alice C. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–351, 1992.

[56] Ram Mohan V. Rachamadugu. A note on the weighted tardiness problem. *Operations Research*, 35(3):450–452, 1987.

[57] M. Schwiegershausen, H. Kropp, and P. Pirsch. A system level HW/SW partitioning and optimization tool. In *Proceedings of European Design Automation Conference*, pages 120–125, September 1996.

[58] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:56–66, 1956.

[59] V.S. Tanaev, V.S. Gordon, and Y.M. Shafransky. *Scheduling Theory. Single-Stage Systems.* Kluwer Academic Publishers, Boston, USA, 1994.

[60] Ari P.J. Vepsalainen. State dependent priority rules for scheduling. Technical Report CMU-RI-TR-84-19, The Robotics Institute, Carnegie-Mellon University, 1984.

[61] Ari P.J. Vepsalainen and Thomas E. Morton. Priority rules for job shops with weighted tardiness costs. *Management Science*, 33(8):1035–1047, August 1987.

[62] A. Volgenant and E. Teerhuis. Improved heuristics for the n-job single-machine weighted tardiness problem. *Computers & Operations Research*, 26:35–44, 1999.

[63] J.K. Weeks. A simulation study of predictable due dates. *Management Science*, 25:363–373, 1979.

[64] T.-Y. Yen. *Hardware-Software Co-Synthesis of Distributed Embedded Systems.* PhD thesis, Princeton University, June 1996.

[65] T.-Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer Aided Design*, pages 288–294, November 1995.

We never know how high we are
Till we are called to rise;
And then, if we are true to plan,
Our statures touch the skies.

The heroism we recite
Would be a daily thing,
Did not ourselves the cubits warp
For fear to be a king.

Emily Dickinson, *We Never Know How High*