Synthesis of Application-Specific Multiprocessor Systems

Shiv Prakash

CENG Technical Report 93-06

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

SYNTHESIS OF APPLICATION-SPECIFIC MULTIPROCESSOR SYSTEMS

by

Shiv Prakash

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

January 1993

Copyright 1993 Shiv Prakash

Dedication

To my wife, and my parents.

Acknowledgements

I would like to take this opportunity to express my gratitude to a number of people who have made this thesis possible. I am indebted to my advisor Prof. Alice Parker for the inspiration and guidance I received from her during my dissertation work. In addition, I would like to thank her for ensuring constant availability of all the necessary resources for research. Finally, she has been a great source of encouragement, help, and friendship. Without her encouragement, I would not have been in a position to write this page.

I also wish to thank Prof. Michel Dubois who gave me useful feedback while this thesis was still forming, and served on my dissertation committee. Thanks are also due to Prof. Shahram Ghandeharizadeh for serving on my dissertation committee.

I would also like to thank Prof. Melvin Breuer for serving on my guidance committee, and for his valuable criticisms and suggestions. Thanks are also due to Profs. George Papavassilopoulos, Kim Korner, and James Yee for serving on my guidance committee. Thanks are also due to Prof. Sarma Sastry for his ready support.

I would like to thank Prof. Lou Hafer of Simon Fraser University for introducing me to Design Automation and some of the techniques used in this dissertation. I gratefully acknowledge several technical discussions held with him during my dissertation research. He was readily available for help. His computer program *Bozo* has been extensively used in this work.

My parents have been a constant source of encouragement. I thank my parents and other family members for their continuous love, support, and sacrifice. I gratefully dedicate this thesis to them.

I especially thank my wife Shakti for her love, support, and patience through the years of graduate school. She has been very understanding and a great source of inspiration during this ordeal. This dissertation is dedicated to her as well.

During my years at USC, I benefited greatly from interacting with many colleagues and friends. I thank my colleagues Kayhan Kucukcakar, Mitch Mlinar, Sally Hayati, Rajiv Jain, Jorge Seidel, Dick Leubben, Jagannath Raghavendran, Pravil Gupta, Atul Ahuja, Chih-Tung Chen, Yung-Hua Hung, Jen-Pin Weng, Charles Njinda, and Esther Brotoatmodjo for their friendship and other help. Thanks are also due to Donnalyn Combest and Mary Zittercob for their help.

Some other friends who helped and cared during this ordeal are Vinay Gupta, Ravi Chennagiri, Amit Majumdar, Deb Mukherje, and Gopal Srinivasan.

Finally, I gratefully acknowledge the financial support from the Department of Air Force, the Department of Army and the Department of Navy under Contract No. N00039-87-C-0194, and the Defense Advanced Research Projects Agency under contract No. JFBI90092. The views and conclusions considered in this dissertation are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Defense Advanced Research Projects Agency or the U.S. Government.

Contents

Dedication ii

Acknowledgements iii

List Of Figures x

List Of Tables xi

Abstract xiii

1 Introduction 1
1.1 Computer-Aided Design of Integrated Systems 1
1.1.1 Non-inferior Designs 2
1.2 High-Level Synthesis 2
1.2.1 A Brief Overview 3
1.2.2 The ADAM System 3
1.2.3 State of High-Level Synthesis Research 4
1.3 General System-Level Synthesis Problem 5
1.4 The USC Project 7
1.5 Multiprocessor Systems 7
1.5.1 Motivation for Application-Specific Multiprocessor Systems 9
1.5.2 Level of Parallelism 10
1.5.3 Heterogeneous Processing 11
1.5.3.1 Motivation for Heterogeneous Processing 11
1.5.3.2 Types of Heterogeneity 12
1.6 The General Design Problem 13
1.7 The SOS Approach 13
1.7.1 The Problem Statement 14
1.7.2 Importance of Work 15
1.7.2.1 Example Applications 16

1.7.3 The Problem Approach 17
1.8 Thesis Organization 18

2 Previous Related Research **19**
2.1 Multiprocessor Research 20
2.1.1 Task Allocation Work 20
2.1.1.1 Graph-Theoretic Approach 20
2.1.1.2 Analytical Modeling Approach 23
2.1.1.3 Mathematical Programming Approach 25
2.1.1.4 Heuristic Approach 27
2.1.2 Mapping of Specialized Algorithms 28
2.1.3 Multiprocessor Synthesis Work 28
2.1.4 Scheduling of Precedence Graphs 29
2.2 Datapath Synthesis Research 29
2.3 Array-Processor Synthesis Research 31
2.4 System-Level CAD Tools 32

3 The SOS Approach **34**
3.1 Overview of the SOS Approach 35
3.2 A Specific SOS Model 38
3.2.1 A Representative Task Model 39
3.2.2 A Representative System Model 40
3.2.3 The Mathematical Programming Model 42
3.2.3.1 The Constraints 42
3.2.3.2 Objective Functions 50
3.2.4 Synthesis Using the Model 53
3.2.4.1 Linearization of the Model 53
3.2.4.2 Size of the Linearized MILP Model 55
3.2.4.3 Solution of the Model 56
3.2.5 Comparison with Hafer's RT-Level Model 56

4 Experiments with Point-to-Point Interconnection Model **58**
4.1 Example 1: Four-Subtask Task Graph 58
4.2 Tradeoff Studies Using the Four-Subtask Graph 63
4.2.1 Experiment 1: Increase the Communication Time 63
4.2.2 Experiment 2: Increase the Execution Time 63
4.3 Some More Tradeoff Studies 65
4.3.1 Variation in Processor Costs 65
4.3.2 Variation in Communication Link Cost/Performance . . 66
4.3.3 Imposition of Arbitrary Constraints 66
4.3.4 New Set of Data 68
4.4 Example 2: Nine-Subtask Task Graph 70

5 Bus-Style Interconnection Model **74**
5.1 The System Model 74
5.2 The Mathematical Programming Model 75
5.2.1 The Constraints 75
5.2.2 Objective Functions 76
5.3 Synthesis Using the Model 76
5.4 Experiments and Results 77
5.4.1 Example 1: Four-Subtask Task Graph 77
5.4.2 Example 2: Nine-Subtask Task Graph 78

6 Memory Modeling and Other Extensions of the SOS Model **80**
6.1 Memory Modeling 80
6.1.1 Local Memory Costs 81
6.1.1.1 The Task Model and the System Model 81
6.1.1.2 The Mathematical Programming Model 82
6.1.1.3 Synthesis Using the Model 83
6.1.2 Experiments and Results 84
6.1.2.1 Example 1: Four-Subtask Task Graph 84
6.1.2.2 Example 2: Nine-Subtask Task Graph 87
6.1.3 Memory Buffers for Communication 90
6.1.3.1 Input Buffer 91
6.1.3.2 Output Buffer 95

7 Making SOS More Practical **99**
7.1 Some Comments on the Practicality of SOS 99
7.2 Runtime Improvement for the MILP Model 100
7.3 Branching Strategies 103
7.3.1 Assignment of Priorities to Binary Variables 104
7.3.1.1 Motivation 104
7.3.1.2 Generalized Design Strategy 105
7.3.1.3 Simplified Strategies 106
7.3.1.4 Experiments with the Simplified Strategies 112
7.3.1.5 Dynamic Priorities 114
7.3.2 Branching on a Group of Binary Variables 116
7.3.2.1 Motivation 116
7.3.2.2 Possible Branching Groups 117
7.3.2.3 Experiments with Branching Groups 118
7.3.2.4 More Branching Groups 121
7.4 Constraint Satisfaction 124
7.4.1 Constraint Satisfaction Techniques 125
7.4.2 Experiments and Results 127

7.4.2.1 Bonsai Alone 127
7.4.2.2 Bonsai Combined with (σ, α, ϕ, rest) Ordering . 128
7.4.2.3 Bonsai Combined with Group Combination 1 . . 129
7.4.2.4 Bonsai Combined with Group Combination 2 . . 130
7.4.2.5 Bonsai Combined with Group Combination 1 & (σ, α, ϕ, rest) Ordering 130
7.5 Some Experiments with Nine-Subtask Task Graph 131
7.5.1 (σ, α, ϕ, rest) Ordering 132
7.5.2 Bonsai with Group Combination 1 & (σ, α, ϕ, rest) Ordering 133
7.5.3 Discussion 134

8 Development of Some Theory and Useful Bounds 135
8.1 Motivation for Computing Bounds 135
8.2 Bounds on Cost and Performance 137
8.2.1 Bounds on the Number of Processors 137
8.2.2 Bounds on the Number of Communication Links 139
8.2.3 Bounds on the Cost of the Synthesized System 139
8.2.4 Bounds on the Performance of the System 140
8.3 Tighter Bounds: Associated Difficulties and Problems 141
8.3.1 Critical Path Estimation 141
8.3.1.1 Critical Path Estimation for "Well-Partitioned Tasks" 150
8.3.1.2 Extension of Al-Mouhamed's Work 151
8.3.2 Estimation of the Upper Bound on the Task Completion Time 152
8.3.3 Estimation of the Upper Bound on the System Cost 154
8.4 Bounds for Special Cases 155

9 Algorithmic/Heuristic Procedures for System-Level Synthesis 161
9.1 Review of Scheduling Heuristics 162
9.2 Extending Heuristics to Heterogeneous Systems 164
9.3 Synthesis of Pipelined Designs 168

10 Conclusion and Future Research 171
10.1 SOS Modeling Approach 172
10.1.1 Contributions 172
10.1.1.1 Use of SOS in Partial Design and Verification . . 176
10.1.2 Future Research 177
10.1.2.1 Enhanced SOS Models 177
10.1.2.2 Other Research Directions 179
10.2 Runtime Improvement for SOS Models 179
10.2.1 Contributions 179

10.2.2 Future Research 180
10.3 Theory and Bounds Development 182
10.3.1 Contributions 182
10.3.2 Future Research 182
10.3.2.1 Consideration of Subtleties 182
10.3.2.2 Extension to More Complex Design Situations . . 183
10.4 Development of Heuristics 183
10.4.1 Contributions 183
10.4.2 Future Research 184

Reference List **184**

Appendix A
The MILP Model for Four-Subtask Task Graph 194
A.1 The Model 194

List Of Figures

1.1 High-Level Synthesis in the ADAM System 4
1.2 The USC (Unified System Construction) Project 8
1.3 SOS Inputs and Outputs 16

3.1 Example 1 Task Graph 36
3.2 Synthesized Multiprocessor System I for Example 1 Graph 37
3.3 Timing Variables 43
3.4 Processor-selection Constraint 44
3.5 Output-availability Constraint 45
3.6 Subtask-execution-start Constraint 46
3.7 Subtask-execution-end Constraint 46
3.8 Data-transfer-start Constraint 47
3.9 Data-transfer-end Constraint 48
3.10 Processor-usage-exclusion Constraint 49
3.11 Communication-link-usage-exclusion Constraint 50
3.12 Communication-link-creation variable 51
3.13 Relationship between β-type and σ-type Variables 52
3.14 Relationship between χ-type and σ-type Variables 53

4.1 Synthesized Multiprocessor System I and Detailed Schedule for Example 1 61
4.2 System I Schedule (Concise) for Example 1 62
4.3 Example 2 Task Graph 71

8.1 Example for Upper Bound on Number of Processors 138
8.2 An Example to Illustrate a Problem with Critical Path Estimation 143
8.3 An Example to Illustrate Another Problem with Critical Path Estimation 145
8.4 The Graph for Theorem 8.3.1.2 147
8.5 A Graph to Demonstrate Yet Another Problem with Bounds . . . 159

List Of Tables

4.1 Processor Characteristics - Example 1 59
4.2 Example 1 Systems 60
4.3 New Processor Characteristics - Example 1 68
4.4 Example 1 Systems with New Data 68
4.5 Processor Characteristics - Example 2 70
4.6 Example 2 Systems 70

5.1 Systems for Four-Subtask Graph (Bus-Style) 77
5.2 Systems for Nine-Subtask Graph (Bus-Style) 78

6.1 Memory Requirements for Four-Subtask Graph 84
6.2 Bus-Style Systems for Four-Subtask Graph (With Memory) 85
6.3 Point-to-Point Systems for Four-Subtask Graph (With Memory) . 86
6.4 Memory Requirements for Nine-Subtask Graph 87
6.5 Bus-Style Systems for Nine-Subtask Graph (With Memory) 88

7.1 Runtime Statistics for No Priority Assignment (Four-Subtask Graph) 113
7.2 Runtime Statistics for (σ, α, ϕ, rest) Ordering (Four-Subtask Graph) 113
7.3 Runtime Statistics for (γ, σ, α, ϕ, rest) Ordering (Four-Subtask Graph) 114
7.4 Runtime Statistics for σ-Group, γ, δ-Group (Four-Subtask Graph) 119
7.5 Runtime Statistics for γ, δ-Group, β, σ-Group (Four-Subtask Graph) 119
7.6 Runtime Statistics for Group Combination 1 & (σ, α, ϕ, rest) Ordering (Four-Subtask Graph) 120
7.7 Runtime Statistics for Group Combination 2 & (σ, α, ϕ, rest) Ordering (Four-Subtask Graph) 121
7.8 Bonsai Runtime Statistics (Four-Subtask Graph) 128
7.9 Bonsai Runtime Statistics for (σ, α, ϕ, rest) Ordering (Four-Subtask Graph) 129
7.10 Bonsai Runtime Statistics for σ-Group, γ, δ-Group (Four-Subtask Graph) 129

7.11 Bonsai Runtime Statistics for γ, δ-Group, β, σ-Group (Four-Subtask Graph) 130
7.12 Bonsai Runtime Statistics for Group Combination 1 & (σ, α, ϕ, rest) Ordering (Four-Subtask Graph) 131
7.13 Runtime Statistics for No Branching Strategy (Nine-Subtask Graph) 132
7.14 Runtime Statistics for (σ, α, ϕ, rest) Ordering (Nine-Subtask Graph) 132
7.15 Bonsai Runtime Statistics for Group Combination 1 & (σ, α, ϕ, rest) Ordering (Nine-Subtask Graph) 133

Abstract

The focus of this thesis is on application-specific multiprocessor systems. Application-specific multiprocessor systems can provide better performance and/or cost-effectiveness, as the application and the system can be very well-matched. The thesis addresses the problem of design of application-specific multiprocessor systems. A synthesis approach has been advocated for the problem. SOS is a formal synthesis approach developed that can be used for design of application-specific multiprocessor systems.

The SOS approach is quite general and flexible. Given the application task characteristics and the system (to be designed) characteristics, the approach involves creation of a formal model using mathematical programming. The formal model encompasses all the relevant tradeoffs that must be considered to synthesize an optimal design. Solution of the mathematical programming model produces an optimal design. Static scheduling and allocation is performed in the SOS approach, in order to provide a better match between the application and the system and thus produce an overall optimal design. SOS synthesizes heterogeneous systems, as they can better match the application and hence tend to be more efficient as well as cost-effective.

The SOS approach has been validated by developing applicable mathematical programming models for different system models and performing synthesis experiments with them. The practicality of the approach has also been addressed, and some techniques for reducing the computer time to solve the SOS mathematical programming model have been investigated. The techniques show significant improvement is achievable through them. An attempt is also made to develop

some theory and bounds with the dual goal of making SOS more practical as well as understanding the synthesis problem better. Some heuristic ideas have been outlined for the synthesis problem, and some scheduling heuristics have been developed for heterogeneous systems. Finally, the subject of "system-level pipelining" is introduced.

Chapter 1

Introduction

1.1 Computer-Aided Design of Integrated Systems

With VLSI systems becoming more and more commonplace, and with myriads of new application avenues opening for VLSI systems, there is a growing need for design of hardware systems for specific applications. In other words, there is a need to design hardware systems that will perform a given class of tasks efficiently. General-purpose systems may not be able to provide the optimal performance and/or cost-effectiveness. More and more special-purpose systems are being constructed now than ever before. Put philosophically, the "from problems to systems" approach is being practiced. These application-specific, special-purpose systems have become so complex that system-level design decisions cannot be made without the aid of computer tools. As system complexity and size have increased, designers have relied increasingly on analysis techniques like simulation and queueing models for assistance during the design process. However, in most cases, system design decisions have been left to the human designer, who often uses a "generate and test" approach to confirm the validity of his or her decisions. There is a growing need to develop CAD tools for system-level design. CAD tools have been successfully used in design of integrated circuits (ICs),

primarily so because the "from problems to systems" approach is often used in design of such circuits. It seems that a similar approach used at the system-level should lend itself well to the usage of CAD tools for design.

1.1.1 Non-inferior Designs

In the "from problems to systems" approach, the system is specifically tuned for a target application. Ideally, one would like to design the "optimal" system which can perform this task efficiently. However, this is easier said than done. The problem faced by system designers is that usually there is a set of design objectives rather than a single objective. Typically, these design objectives are in conflict with each other. Therefore, instead of an absolute optimal design we have a set of *non-inferior* designs. A non-inferior design is a system which can be improved with respect to any one objective only at the expense of others. The non-inferior designs reflect the tradeoffs among the design goals. Thus to find the best solution, the designer may have to explore more than one non-inferior design. This is where computer-aided design (or design automation) can be helpful. In fact, an efficient CAD tool can generate the whole set of non-inferior designs and then the system designer can select the design which embodies the most acceptable tradeoff among the various design objectives.

1.2 High-Level Synthesis

Computer-aided design has been successfully used for design of ICs. In fact, it has been used for automatic synthesis of application-specific integrated circuits (ASICs). Given an algorithmic-level behavioral specification of a digital system, the register-transfer design for a single ASIC is synthesized to realize the behavior. Such a synthesis is known as *high-level synthesis* [MPC90].

1.2.1 A Brief Overview

In general terms, high-level synthesis accepts the following inputs:

- an algorithmic-level behavioral specification of a digital system, and
- a set of constraints and goals to be satisfied.

It produces the following output:

- a register-transfer level (RTL) structure that realizes the given behavior and best meets the constraints and goals.

In the above, the behavioral specification essentially reflects the mapping from inputs to outputs. Constraints usually deal with design parameters such as performance, area of the ASIC, or power consumption in the ASIC. As an example, the design goal could be to minimize area of the ASIC while achieving a certain required performance. The RTL structure consists of a network of registers, functional units (operators; e.g., adder, multiplier), multiplexors, and buses. The key point is that usually there are many different RTL structures that can be used to realize a given behavior; and the goal of high-level synthesis is to find the one that best meets the design constraints and goals.

1.2.2 The ADAM System

A high-level synthesis system known as the *Advanced Design Automation System* (ADAM) has been developed at the University of Southern California [PHJ+88]. The ADAM synthesis system accepts behavioral input specified either in the VHDL hardware description language or in terms of a data-flow graph where nodes of the graph represent operations (e.g., add, multiply) and arcs represent both the values transferred between nodes and the precedence constraints between operations. In addition, ADAM also requires as input a design library of available hardware operators (e.g., adders, registers), and their cost/performance characteristics. The constraints and goals to be satisfied are specified in terms of

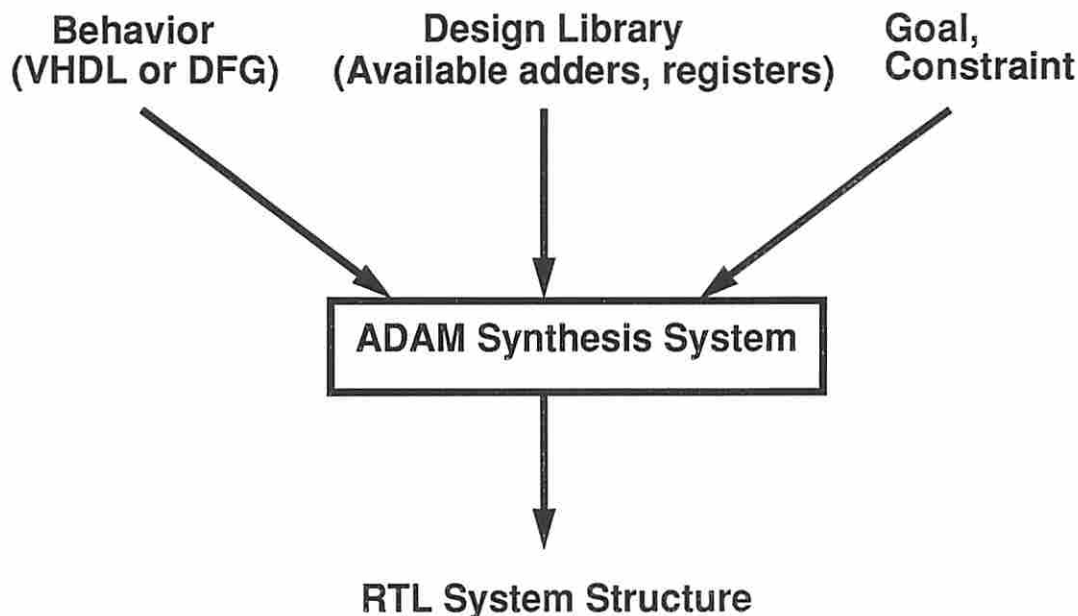


Figure 1.1: High-Level Synthesis in the ADAM System

area and/or performance of the ASIC desired. The synthesis system attempts to produce an RTL structure for the ASIC that realizes the input behavior and best meets the specified constraints/goals. The RTL structure consists of the components specified in the design library. Figure 1.1 gives an overview of ADAM input/output for high-level synthesis.

1.2.3 State of High-Level Synthesis Research

The field of high-level synthesis has evolved significantly during the last decade, and is considered a fairly mature field [MPC90]. Its future research directions are geared towards system-level issues instead of single ASIC designs. In the future, for example, it would be desirable to have a high-level synthesis system system that can deal with issues relating to the number and configuration of system-level components such as memories, processors, and controllers, instead of concentrating on how these individual pieces are designed. Not much research

has been done to provide design aids at this system level. To summarize, the future of high-level synthesis is *system-level synthesis.*

1.3 General System-Level Synthesis Problem

Now, the question is "what exactly is system-level synthesis?" Even though the answer is not very clear, we know while high-level synthesis deals with RTL components, system-level synthesis must concern itself with system-level components. It should focus on decisions like the number of system-level components to be included in the system, and how to configure and interconnect them in the system. How exactly the individual components are designed is not a primary concern at this level. The next logical question is "what is a system-level component?" Again, the answer is not very clear and depends on the application under consideration. Also, to some extent, the answer depends on the designer. Whatever options the designer wishes to consider can be included in the set of system-level components available. Here are some typical system-level components:

- Processors: These could be general-purpose or special-purpose.
  - General-purpose: For example, different kinds of microprocessors (Motorola 6809, 68008, 68010, and Intel 80386)
  - Special-purpose: These are processors that are designed either to perform some specialized functions (to achieve the required functionality) or to improve performance for certain specific functions. Some examples with obvious meanings are a linear algebra module, a numerical methods module, a floating-point module, a string-processing module. These could include filters in digital signal processing (DSP) applications. Another example would be specialized components for system reliability, such as Hamming encoders/decoders.

- Memories: RAM, ROM, or cache memories; and memory management modules (components taking care of memory management).

- Peripheral devices: For example, I/O modules, disk, tape and other resources

- Bus interfaces: A variety of bus interfaces may be available; e.g., AT bus.

The above list is by no means comprehensive. In general, system-level synthesis should be able to deal with such components and any other components dictated by the application and the designer.

Now, following the "from problems to systems" philosophy and extending the ideas from high-level synthesis, the general system-level synthesis problem can be loosely defined as follows. Given an application task, a set of system-level components, and a set of constraints/goals (say in terms of cost and performance of the system), configure a system consisting of components chosen from the given set such that the system is capable of performing the given task and meets the constraints/goals. Obviously, some sort of "optimal" system should be synthesized. In fact, one would like a non-inferior system which best meets the constraints/goals. This is definitely one of the motivating factors for system-level synthesis.

It is easy to see that system-level synthesis is a very hard problem, especially if one is looking for an optimal or non-inferior system. Several tradeoffs are involved. For example, hardware/software tradeoffs may have to be considered. If a part of the computation (let us refer to it as a *subtask*) can be performed on a specialized processor as well as a micro-processor in the given set of components, then which processor type should be chosen? A specialized processor may enhance the performance, but it may be more expensive; obviously there is a tradeoff then. All such tradeoffs need to be investigated in order to come up with an optimal answer. Another reason why the problem is hard is because the system-level design process itself is not a formalized one and definitely not

well understood. How a human designer performs system-level design is not very clear. That means we do not have an algorithm which can be emulated in a computer tool to perform synthesis. It is the opinion of the author that the general system-level synthesis problem is an NP-hard problem.

1.4 The USC Project

The goal of the present research at the University of Southern California is to extend the ADAM system to perform "higher-level" or system-level synthesis where a system is definitely something more complex than a single ASIC. The new effort is called the Unified System Construction (USC) Project [PKPW91].

Figure 1.2 illustrates the overall structure of the USC project. The USC project is an effort to consider some system-level design considerations. Other parts of the project are not relevant to the subject of this thesis, and will not be discussed here. The only relevant part is the SOS component which deals with synthesis of multiprocessor systems. The focus of high-level synthesis is on a single ASIC. However, we believe that it is important to be able to synthesize multiprocessor systems. The SOS (Synthesis Of Systems) project is an effort in this direction and is the subject of this thesis. SOS synthesizes multiprocessor systems based on a Multiple-Instruction Multiple-Data (MIMD) architecture.

1.5 Multiprocessor Systems

In general, a multiprocessor system consists of several processors interconnected for communication through some mechanism. Usually, the communication between processors is asynchronous in nature in MIMD systems. MIMD systems are suitable for a large class of applications because of their flexibility.

With the advent of VLSI technology, several low-cost micro-processors and other kinds of processors have appeared. This has made multiprocessor systems economically viable in many situations. Several such systems have been designed

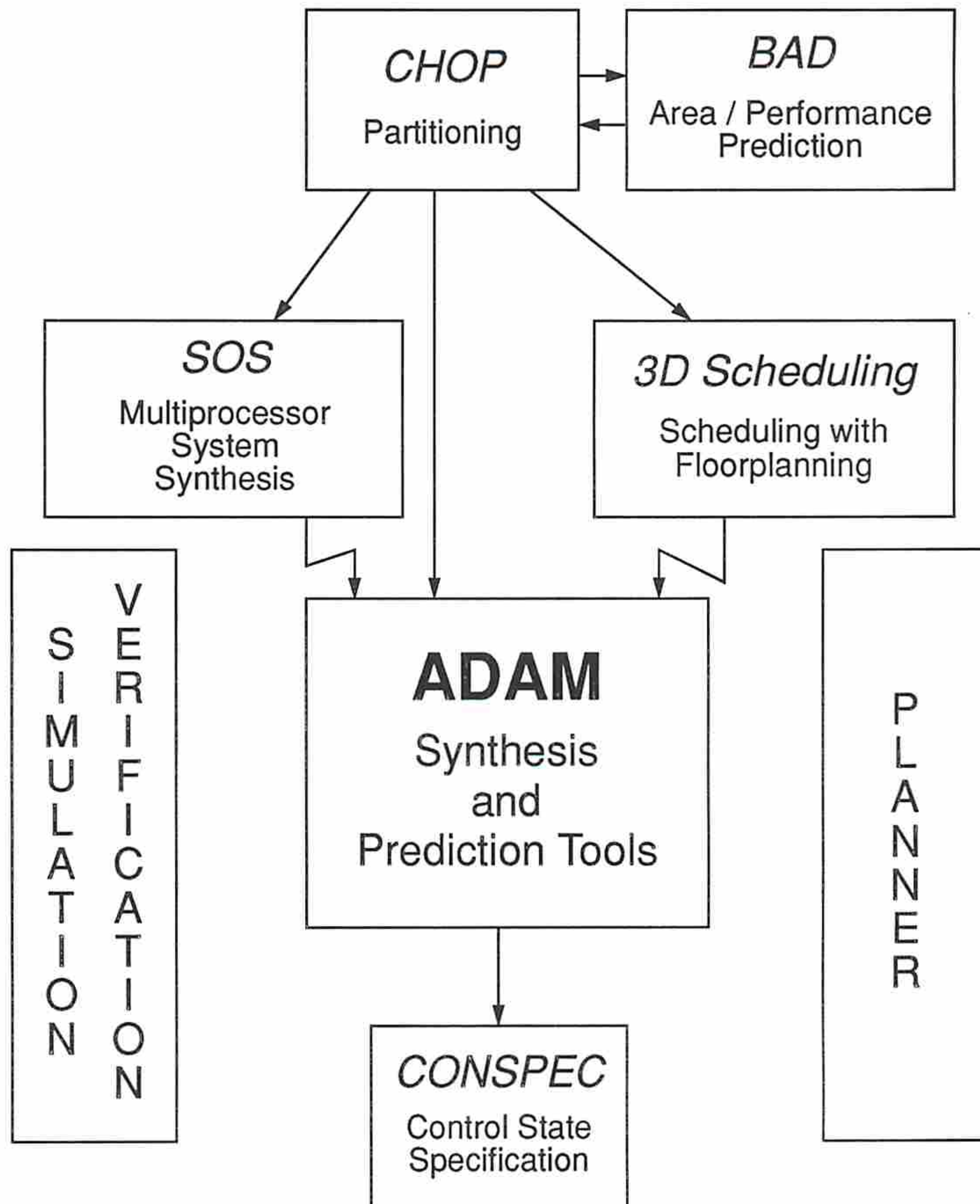


Figure 1.2: The USC (Unified System Construction) Project

and implemented in recent years. The modularity, flexibility, and reliability of multiprocessor systems make them very attractive to many types of applications.

Of course, there are several problems and issues that need to be addressed for multiprocessor systems to be effective. The well-known saturation effect, where the system performance does not increase linearly and in fact eventually starts to decrease as we increase the number of processors in the system, requires that such systems be designed very carefully. Several factors need to be considered: 1) interprocessor communication (IPC) which leads to the saturation effect, and 2) the computation load on each processor. These two factors conflict. The second factor advocates a larger number of processors in order to increase the degree of parallel processing in the system. However, the first factor argues for a lower number of processors in order to minimize the communication cost. Thus, multiprocessor system design has to consider all such tradeoffs.

1.5.1 Motivation for Application-Specific Multiprocessor Systems

In order to design an efficient multiprocessor system, we have to consider factors such as IPC and the computation load on processors. However, it is difficult to know the magnitude of such factors in advance unless the application is well defined and the system is targeted towards this specific application. Efficient general-purpose multiprocessor systems are harder to design for this reason. However, the emergence of several applications for multiprocessor systems has led to usage of dedicated systems in many situations and made the same desirable in many other situations in order to improve the performance. When designing a system for a specific application, the system structure can be closely matched to the problem structure to improve performance. With this in mind, this thesis focuses on application-specific multiprocessor systems.

There are several real-time DSP applications which are fairly deterministic in nature. Dedicated multiprocessors are used and the same set of computations is

repeatedly performed. Generally, one wants low-cost systems with high performance to meet real-time constraints in these applications. For such applications, general-purpose systems are not very useful. In fact, several micro-processors are available for DSP applications (e.g., Motorola DSP56000, DSP96002, and TI signal processors), and DSP multiprocessor systems are designed using them. The deterministic nature of the applications allows design of some very efficient systems.

In a deterministic application, the overall task (let it be divided into several subtasks) to be performed can be well-matched to the system being designed. This is because the subtasks can be allocated to the processors in the system at design time. In fact, we can use static scheduling techniques to schedule the task onto the system at design time. Use of static scheduling and allocation leads to a significant savings in hardware/software at run time. Thus, cost-effective and efficient systems result. Use of static scheduling and allocation for deterministic applications allows for design of some efficient specialized multiprocessor architectures. In [LB90], Lee and Bier describe a class of specialized multiprocessor architectures for real-time DSP. Efficient systems based on such architectures can be synthesized.

Since the design of application-specific multiprocessor systems follows the "from problems to systems" paradigm, it is amenable to CAD in general and synthesis in particular.

1.5.2 Level of Parallelism

When designing an application-specific multiprocessor system, one of the things to base the design on is the amount of parallelism present in the given application.

In general, for a given application task, parallelism could be exploited at different levels. The task could be partitioned into smaller subtasks. Now, parallelism could be exploited among the subtasks as well as within each subtask. In

general, one could have several levels of parallelism: e.g, job, task, process, variable and bit levels. Parallelism among the subtasks may constitute task/process-level parallelism. Parallelism within each subtask may consist of variable-level parallelism. In this thesis, we focus on parallelism among major subtasks. At this level, graph representation of the task (i.e., a graph where nodes are the subtasks) tends to be irregular. We cannot expect a task's major subtasks to be identical or similar. Often a task contains a mix of quite different subtasks with quite different processing requirements. In such cases, a heterogeneous multiprocessor system appears more suitable for parallel processing.

1.5.3 Heterogeneous Processing

Heterogeneous processing is the use of several different types of processors, processing components, and/or connectivity paradigms to optimize performance and/or cost-effectiveness of the system. For example, different processor/processing component types could include vector processors, SIMD processors, MIMD processors, special purpose processors, and data-flow processors. Similarly, different connectivity paradigms could include bus, point-to-point, ring, or a mixture of these. The Purdue mixed-mode PASM system [SSKD87] is an example of a system using heterogeneous processing.

1.5.3.1 Motivation for Heterogeneous Processing

The motivation for using heterogeneous processing is well documented by Freund and Conwell [FC90]. Essentially, Amdahl's Law [Amd67] lies at the core of the need for heterogeneous processing.

Let us assume that the given application task consists of several diverse subtasks that need to be performed in order to accomplish the overall task. Amdahl's Law states that any single type of super-speed processor used on such a heterogeneous task often spends most of its time on the subtasks it does poorest. So, if we choose any single processor type, it would perform the subtasks suited for the

type very quickly and may not perform the remaining subtasks so quickly. One approach to overcoming this effect of Amdahl's Law is to have a heterogeneous system consisting of several processors of different types that match the computation requirements of different subtasks. Heterogeneous systems are useful when the overall performance is crucial, since in a heterogeneous system, individual subtasks could be matched to best-suited processor types and such a matching leads to improved performance of the system.

1.5.3.2 Types of Heterogeneity

Conceptually, we consider two forms of heterogeneity:

- Processor heterogeneity: Here there are two types.
  - Functionality heterogeneity: This refers to heterogeneity in terms of the functionality of the processors. Two processors could be different in terms of the types of subtasks they would be capable of performing. This notion allows inclusion of special-purpose processors in the system.
  - Cost-speed heterogeneity: This refers to heterogeneity in terms of the cost-speed characteristics of the processors. Two processors could be capable of performing a given subtask, but the speeds of execution would be different.
- Interconnection heterogeneity: We would like to allow different styles of interconnection. Not all processors in the system have to be interconnected in the same way; i.e., there is not necessarily a uniform structure to the whole system. Particularly, we would consider point-to-point interconnection between processors. In general, point-to-point interconnection can lead to improved performance.

1.6 The General Design Problem

The general problem is: given an application, design a multiprocessor system that is well-suited for the application and also meets the specified set of constraints/goals.

The general problem involves the following steps:

- Come up with a precise description of the application: specify the application as a computation task, partition the task into subtasks, and specify the interaction between subtasks. A possible specification format here is to use a graph representation.

- Identify a list of components (along with relevant characteristics) that can be used in the system design (for example, processors, interconnection structures). A relevant characteristics could be the cost of a given processor.

- Determine how effective each component is for the application; i.e, how effective a given processor is for a given subtask, or how effective a given point-to-point communication link is in taking care of IPC.

- Assemble a mix of components that best handles the given task and also satisfies any other design constraints/goals.

The primary focus of this thesis is on the last step. Indeed, the SOS approach is geared towards dealing with the "assembling" aspect of the system design.

1.7 The SOS Approach

"SOS" is an acronym for "Synthesis Of Systems." It is an approach for synthesis of application-specific multiprocessor systems. Below, we identify the exact design problem solved by SOS.

1.7.1 The Problem Statement

Conceptually, the problem handled by SOS can be specified in terms of the following inputs/outputs.

- The inputs are
  - a behavioral specification of an application-specific task (as a set of communicating subtasks),
  - a library of components to be used in system design, and
  - a set of constraints and goals to be satisfied.
- The output is
  - a multiprocessor system which can implement the given task behavior and best meets the constraints and goals specified.

The task specification is in terms of a graph where nodes represent subtasks and arcs represent both data transfer and precedence between the subtasks. The library of components depends on the type of system desired. For example, it could consist of processors and interconnection structures. Constraints and goals are in terms of cost and performance of the system to be designed. The multiprocessor system designed essentially consists of a set of components selected from the library. If the library consists of processors and interconnection structures, then the system would consist of a set of selected processors and interconnection between them.

As one can see, the focus of SOS is on system design. However, in order to optimize overall performance and/or cost-effectiveness of the system being designed, it is imperative to consider the issue of usage of the system to implement the given task concurrently with decisions for design. One of the prime concerns relating to the effective usage of the system is the issue of scheduling/mapping the application task onto the system. As we mentioned in Section 1.5.1, static scheduling and allocation should be used at design time for application-specific

multiprocessor systems, as it leads to more cost-effective and efficient systems. Indeed, the SOS approach is scheduling-driven in the sense that it deals with scheduling/mapping as a primary issue and performs static scheduling and allocation at design time. Because of this aspect of the SOS approach, it also produces the following two outputs:

- assignment of subtasks and data transfers to the components in the designed system, and
- task execution schedule.

Thus, SOS produces a custom multiprocessor system, assigns the subtasks and data-transfers to the system and provides a static schedule for the task execution. Figure 1.3 provides an overview of SOS inputs and outputs. It is easy to see that if an optimal or non-inferior system is desired, it is computationally a hard problem. Once again, it is the opinion of the author that the specific system-level design problem addressed by SOS is an NP-hard problem (though a rigorous proof has not been accomplished).

1.7.2 Importance of Work

The work on the problem described in the previous section is significant as it will lead to the following general benefits:

- As the SOS approach attempts to find optimal or non-inferior systems, it is conceivable that some designs which a human designer would not have been able to find could be found using SOS. This means more cost-effective, efficient, and complex systems could be designed.
- As we mentioned earlier, the system-level design process is not very well understood. It is possible that a tool like SOS could provide some insight into the process, and we could understand system-level design better.

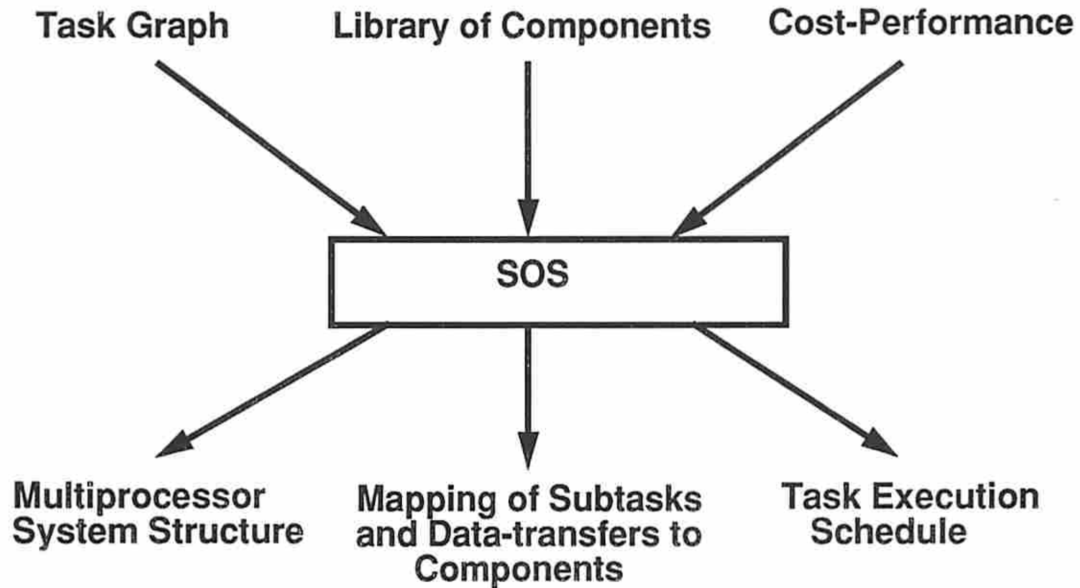


Figure 1.3: SOS Inputs and Outputs

Of course, the primary goal of the research described here is to have a capability to design multiprocessor systems for specific applications. There are several applications where the SOS capability would be useful.

1.7.2.1 Example Applications

Inherently, SOS is not restricted to any specific application domain. The approach itself is fairly general and can be applied to any domain as long as it fits within the general framework of SOS inputs and outputs. Of course, as we mentioned in Section 1.5.2, it is geared towards exploiting parallelism among major subtasks of the application. As major subtasks tend to be non-identical and dissimilar, SOS's primary strength is in dealing with heterogeneous or irregular task graphs. For such graphs, heterogeneous systems are more desirable and indeed SOS can design such systems. This is not to say that SOS cannot design homogeneous systems, but it is not specifically designed for this.

With this in mind, let us look at some example domains where SOS can be used. Of course, SOS's full strength will be utilized only in domains where applications contain multiple diverse subtasks.

The Real-Time DSP Domain: Here several applications are possible:

- Video and image processing applications: Such applications, for example, would arise in a digital TV system. An example of such an application is a fast-Fourier transform computation.

- Audio processing applications: Such applications, for example, would arise in a compact disc system.

- Speech processing applications: An example of such an application is pitch extraction.

Power Systems Domain: Power system applications contain a wide variety of diverse computations. One such example is the "optimum power flow problem" [MT82]. Solution of this problem involves several different types of computations; e.g., matrix operations like LU factorization, sorting, and solution of a set of linear equations.

Robotics Domain: An example application here is a robot arm controller.

Control Systems Domain: Some example applications could be found in the control systems used in automobiles. For example, General Motors uses some multiprocessor-based control systems. It uses Motorola micro-processors to build the systems.

1.7.3 The Problem Approach

As one can see, the approach adopted for design of application-specific multiprocessor systems is one of "synthesis." It involves development of a formal model

for the synthesis problem. Mathematical programming is used to develop such a model. The model depends on the characteristics of the tasks and the systems being considered. The model captures task and system characteristics along with the design rules to be followed. Solution of such a mathematical programming model produces the synthesized design.

1.8 Thesis Organization

The organization of this thesis is as follows. Chapter 2 surveys the related work. Primarily, it describes the multiprocessor task allocation research, datapath synthesis research (particularly the work where mathematical programming is used), some array-processor synthesis work, and some system-level CAD tools. Chapter 3 describes the SOS approach in detail, and gives a complete mathematical programming model for point-to-point interconnection systems. Chapter 4 describes some examples and synthesis experiments using the SOS model for point-to-point interconnection systems. Chapter 5 describes an SOS model for bus-style interconnection systems and some experiments with the same. Chapter 6 shows how to extend SOS models to model memory as an explicit design parameter, and presents some experiments with the extended models. Chapter 7 focuses on runtime issues for the SOS model, and describes some techniques for improving the runtime along with experimental results. Chapter 8 attempts to develop some theory to provide useful bounds on the design parameters. Chapter 9 outlines some heuristic ideas for the synthesis problem. Finally, Chapter 10 summarizes the thesis and outlines future research directions.

Chapter 2

Previous Related Research

The goal of the research described in this thesis is to develop tools and techniques for the design of application-specific systems (multiprocessor systems in this case), or in short, the goal is to perform *system-level synthesis*. Synthesis methodologies for application-specific systems are not well researched yet. The related past research of others covers a broad range of topics. The topics can be grouped into four categories:

- Multiprocessor research
- Datapath synthesis research
- Array-processor synthesis research
- System-level CAD tools

Our research builds on the past multiprocessor research and data path synthesis research.

2.1 Multiprocessor Research

Most multiprocessor design and theory is aimed at general-purpose computing. So, most of the previous related work on multiprocessors is directed at the problem of task allocation for a given architecture. Research has also been conducted in the areas of mapping specialized algorithms and scheduling precedence graphs onto multiprocessors. One research effort by Talukdar and Mehrotra [MT82, MT84] is oriented towards synthesis of multiprocessors.

2.1.1 Task Allocation Work

Several variants of the task allocation problem have been considered and several different approaches have been investigated. We briefly review these efforts next.

2.1.1.1 Graph-Theoretic Approach

Serially Partitioned Tasks: Graph-theoretic techniques have been researched for serially partitioned tasks, i.e, even though there are m subtasks, only one is active at one time. One such effort was Stone's work on the two-processor problem [Sto77]. The work concentrates on the assignment of the subtasks to a system consisting of two processors (single-host, single-satellite) so as to avoid excessive interprocessor communication (IPC) while taking advantage of specific efficiencies of the processors. The goal is to minimize the collective costs due to IPC and to computation. The problem is solved efficiently using the network flow approach. The method involves the construction of a network flow graph in which nodes represent the subtasks to be assigned and edge capacities represent computation and communication costs in such a fashion that the minimum weight cut separating two distinguished nodes in the graph corresponds to the optimal assignment. An effort is also made to extend the method to three and n-processor cases, with only partial success.

In later research [Sto78], Stone reconsidered the two-processor problem and extended the network flow techniques to examine the sequence of optimal assignments found as the load on one processor is held fixed and the load on the other is varied. The research indicated that for every subtask S (in the overall task) there exists a critical load factor f_S such that when the load on the processor with variable load is below f_S, S is assigned to that processor by an optimal assignment, and is otherwise assigned to the other processor. In other words, as the load on the processor with variable load increases, the optimal assignment is always such that subtasks move away from this processor to the other. Thus, successive optimal assignments for successively increasing loads are nested inside each other. In another research project [Gus83], Gusfield developed a parametric computing method for combinatorial problems and applied it to the two-processor problem in the face of varying load levels on both the processors.

Several variants of Stone's two-processor problem have been researched [RSH79, Bok79, SB78]. In [RSH79], Stone considers minimum cost assignment for the case where one processor has limited memory capacity. In [Bok79], Bokhari considers the problem of finding an optimal dynamic assignment of a task. The cost of dynamically reassigning a subtask from from one processor to the other and the cost of subtask residence without execution are included in the model, and network flow algorithms are used. In [SB78], the basic network flow algorithm is generalized to three-processor systems. Static as well as dynamic assignments are considered. An attempt is also made to solve the problem for many-processor systems when the task structure is tree-like. The problem of assignment for tree-like structured tasks is also solved in [Bok81], using an efficient dynamic programming approach. The *shortest tree algorithm* takes into account the interconnection structure of the system (i.e., the speeds of links between pairs of processors), and works by constructing a weighted, layered assignment graph and finding a minimum *sum* weight path in it. Price and Pooch [PP82] extend

the solution technique for tree-like structured tasks to allow an arbitrary subtask intercommunication pattern. The problem is modeled as a directed acyclic search graph and a shortest path algorithm is given that produces an assignment of subtasks to processors. However, the algorithm does not guarantee an optimal solution. A branch-and-bound algorithm is also described which may be applied to the search graph to produce assignments of generally inferior quality but with considerably less computational effort.

Parallel Tasks: In [Bok88, Bok87], Bokhari extends the prior research for serially partitioned tasks [Bok79, SB78, Bok81] to parallel tasks by explicitly taking concurrency into account. A *sum-bottleneck path (SBP) algorithm* is developed that permits the efficient solution of many variants of the problem under some constraints on the structure of the partitions. Point-to-point interconnection is assumed. The system under consideration is of single-host, multiple-satellite type. The problem of partitioning multiple chain-structured parallel programs to minimize the time for execution is solved optimally by constructing an assignment graph and applying the SBP algorithm, under the constraint that each chain is partitioned into two contiguous subchains. The problem of partitioning multiple arbitrarily structured serial programs is transformed into the problem of partitioning multiple chains by use of Stone's network flow algorithm [Sto77] for single-host, single-satellite assignments combined with his results on nested assignments [Sto78]. The problem of partitioning single-tree structured parallel programs is solved optimally by the SBP algorithm, under the constraints that all satellites are identical and that individual maximal subtrees of the given tree are assigned to each satellite. Finally, the problem of partitioning chain-structured parallel programs across chain-connected systems is solved under the constraint that the chain is partitioned into contiguous subchains; and the solution technique is somewhat similar to the technique described in [Bok81], except that a minimum bottleneck weight path instead of a sum weight path yields the optimal

solution. This problem is also considered in [ISB86], where the solution technique is evaluated and an alternative greedy approximation algorithm is described.

Shen and Tsai [ST85] model the task assignment problem as a graph matching problem. The cost function used is the maximum time for a task to complete subtask execution and communication in all the processors. The *minimax* optimization criterion is employed, which implies that the load on the bottleneck processor is minimized instead of the sum of all the processor loads and thus ensures load-balancing. The proposed approach allows consideration of various characteristics of the given system. Graphs are used to represent the subtask relationship of the given task and the processor structure of the distributed system. Subtask assignment to system processors is transformed into a type of graph matching. Although the problem is modeled as graph matching, the solution is not obtained using graph-theoretic algorithms. Instead, the search of optimal task assignment (minimum-cost graph matching) is formulated as a state-space search problem which is solved by the well-known A^* algorithm in artificial intelligence [Nil71].

The limitation of the graph-theoretic approach is that it is not easily extendable to solve the general problem of task allocation for an arbitrary number of processors. Another limitation is encountered in handling restrictions on resources (e.g., memory size restricted) and arbitrary constraints (e.g., constraint on task completion time).

2.1.1.2 Analytical Modeling Approach

In [ISXC86, Sto87], Stone et al. use an analytical approach for modeling and optimization of multiprocessing execution time for random-graph models of programs. An optimal task-assignment policy is derived by optimizing the execution time. The execution time is modeled as the sum of the execution time of the busiest processor and the total communications overhead. The derived policy essentially says that the optimal task assignments are extremal in the sense that

the cost of processing the subtasks is distributed among all processors as evenly as possible or not distributed at all, depending upon the ratio of runtimes to communication times as well as the ratios of the processing speeds of the processors. The policy holds in the case of homogeneous as well as heterogeneous (only in terms of speed, not functionality) processors. Nicol [Nic89] describes similar research for partitioning of random programs in the two-processor case. This result, though striking, has its limitations because of the underlying assumptions made in its derivation. The model of the system assumes that there is an aggregate communication bandwidth among processors, and the total communications overhead is estimated by aggregating the traffic involved. Such a model is reasonable for systems sharing a common bus, but less acceptable for systems containing internal point-to-point connections, as the aggregation tends to obscure nonuniform traffic patterns in a point-to-point communication structure and does not account for saturation of individual point-to-point links. Another limitation arises due to the fact that a random graph is not an accurate model of programs and that the effect of precedence relations is ignored. Another severe limitation can be attributed to the fact that the communications overhead is simply added to the execution time of the busiest processor and thus the execution time formulation does not account for parallelism between communication and task execution. Finally, the optimization process used is based on certain approximations and is not exact. Haddad [Had89a] uses a different approach for modeling of the execution time, which formulates a more accurate analytical representation. In this work, the internal inter-subtask communication times (time spent in communication between two subtasks assigned to the same processor) are also considered. However, precedence relations are ignored here also. The system consists of P heterogeneous processors and L communication channels. A result similar to Stone's policy of "even distribution or no distribution" is obtained by Haddad also under some simplifying assumptions. One of the assumptions used is that the communication channels never get saturated. Although the

results obtained by Stone and Haddad are based on certain limiting assumptions, they may still provide heuristics for our use in the application-specific domain.

2.1.1.3 Mathematical Programming Approach

Integer 0-1 Programming Approach: Chu et al. [CHLE80] considered the problem of optimal allocation (minimum overhead due to IPC) of a set of m subtasks to a set of p (fixed) processors already interconnected in some fashion, and suggested that an integer 0-1 programming approach is most likely to be useful in solving realistic task allocation problems. In this paper, the objective function for the integer 0-1 programming model is formulated as the sum of the processing costs of the subtasks on the processors assigned to them and the total IPC cost. Constraints are formulated to reflect memory size restrictions and task completion time requirements. It is a flexible technique because it allows constraints to be introduced into the model as appropriate to the application.

Ma et al. [MLT82] also report an integer programming model for task allocation. Some extra constraints have been incorporated into the model to meet various application requirements. A branch-and-bound method is used to solve the model. For a given distributed system and a set of constraints, it generates the minimum-cost allocation.

However, the models described in [CHLE80, MLT82] do not consider the effects of precedence relations in the data flow among the subtasks. We are proposing a similar approach for synthesis and we intend to take into account the precedence relations.

Nonlinear Programming Approach for Arbitrarily Partitionable Tasks: Nonlinear programming has been used for task allocation under an assumption that the given task can be split into arbitrary size subtasks. Agrawal and Jagadish [AJ88] present one such model that can be used for an optimal partitioning of the class of computations that are organizable as a one-level tree, and are

homogeneous and separable. The system model consists of a master processor and several similar slave processors. The task is assumed to be divisible into an initial phase, a separable phase and a final phase. The initial and final phases are executed on the master alone. The separable phase can be split into several independent subtasks executing in parallel on slave processors. The separable phase subtasks communicate with the initial phase subtask and the final phase subtask. Given a number of processors n, the goal is to minimize the total execution time (including communications overhead) by partitioning the separable phase into n independent arbitrary size subtasks. Point-to-point communication over the network is assumed. A nonlinear programming formulation is given, which in special cases becomes linear. An iterative technique is also outlined to determine the optimal number of slave processors. One of the major drawbacks with this work is the assumption that continuous partitioning of the separable phase is possible with no communication between the subtasks.

In [Had89b, Had89c], Haddad also assumes that the overall given task can be split arbitrarily. The problem considered in this work is that of minimizing the execution completion time of a given task by partitioning into interacting subtasks and allocating to run on a heterogeneous system, and it is also formulated as a nonlinear programming problem. An exact solution to the problem is given using a new technique, and a theorem is presented stating the necessary and sufficient condition for minimum execution time. The modeling of the execution time is similar in precision to that in [Had89a], in the sense that internal inter-subtask communication times are considered and that precedence relations are ignored.

A limitation of the research described in [AJ88, Had89b, Had89c] is that it is not directly applicable to practical situations where partitioning can usually be done only at specific points.

2.1.1.4 Heuristic Approach

Heuristic methods aim only to find a suboptimal assignment for a task and are usually based on some simplifying assumptions. They are useful when it is important to reduce the amount of computation. A heuristic approach is described in [Efe82], which attempts to minimize IPC by using a clustering approach under a load-balancing constraint. The load-balancing constraint is achieved by balancing the time needed to execute subtasks assigned to processors within a given tolerance.

In [CL87], Chu et al. consider the precedence relationship (PR) among subtasks explicitly and study its effects on the performance. The research indicates the subtask-size ratio between two consecutive subtasks plays an important role in determining whether they should be colocated. A heuristic algorithm considering PR, execution time of subtasks, and IPC is given which attempts to minimize the bottleneck-processor utilization. The algorithm works by grouping the subtasks based on IPC, PR effects, and size of the group, and is shown to generate better task assignments than those not considering the PR effects. Only homogeneous systems with a given number of processors are considered. Constraints on task completion time are given a consideration. Houstis [Hou90] also discusses similar issues and describes heuristic algorithms for task allocation to homogeneous bus connected systems. The objective is to minimize the total processing time of the task. The algorithms try to minimize IPC delays (computed by taking interconnection network characteristics into account) by a clustering approach while keeping PR in mind, under a load-balancing constraint which is achieved by constraining each processor's utilization not to exceed a prespecified upper bound. An iterative algorithm is also given to determine the optimal number of processors. This is the first attempt at solving the allocation problem to determine the optimal number of processors. This research also considers data to memory assignment.

2.1.2 Mapping of Specialized Algorithms

There have been research efforts directed towards the mapping of specialized signal processing algorithms onto multiprocessors. For example, an early study mapping algorithms such as the Fast Fourier Transform onto the CM* multiprocessor in order to meet real time constraints was undertaken by Brantley [Bra79]. Also, there have been many research efforts on static scheduling of DSP algorithms on already designed synchronous general purpose multiprocessors (e.g., [BS84]).

2.1.3 Multiprocessor Synthesis Work

As the reader would notice, all the research efforts described in Sections 2.1.1 and 2.1.2 essentially concentrate on the problem of efficient allocation of tasks to given systems with the goal of optimizing the performance. The motivation in these efforts is to efficiently utilize given systems. No effort is directed towards the synthesis of systems, and hence cost of the system as such is not a consideration. In synthesis, one would like to design cost-effective systems that would provide the desired performance for given tasks. One such effort has been described by Talukdar and Mehrotra [MT82, MT84]. This work describes a procedure for high-level synthesis of special-purpose dedicated heterogeneous (only in terms of speed) multiprocessor systems. Precedence relations and cost of the system are given explicit consideration. The goal is to find a minimum execution time system which meets the system cost constraint. The problem is modeled using mathematical programming, though the solution procedure is heuristic and iterative. The core of the solution procedure consists of an interactive program that estimates the minimum execution time of the task for a given system. In this work, no explicit consideration is given to the delays and costs associated with the communication links. Our proposed research intends to model the communication links explicitly.

2.1.4 Scheduling of Precedence Graphs

Research efforts have also been directed to study the problem of scheduling precedence graphs onto homogeneous systems. Fernandez and Bussell [FB73] propose a lower and an upper bound on the number of processors required to execute the graph in a time not exceeding the length of the critical path. They also determine a lower bound on the execution time for a given number of processors. Kasahara and Narita [KN84] describe heuristic algorithms, combining critical path ideas with branch-and-bound, for scheduling to minimize the execution time. However, both the efforts completely ignore the communication overhead. Al-Mouhamed [AM90] describes research which considers the communication overhead. He proposes an approximate lower bound on the completion time, and approximate lower bounds on the number of processors and the number of communication links required to process the graph within this completion time. An approximate lower bound on the completion time is also estimated for a given number of processors. These bounds are estimated by defining the notions of "earliest possible starting time" and "largest possible delay without increasing the completion time" for each of the subtasks in the graph. Since this research explicitly deals with precedence as well as communication, we propose to investigate if it could be extended for heterogeneous systems to estimate bounds that could be used in conjunction with the mathematical programming approach being adopted.

2.2 Datapath Synthesis Research

We propose to use a mathematical programming approach for the multiprocessor synthesis problem. Such an approach has been used for the data path synthesis problem. Hafer and Parker [HP83, Haf81] have used a mixed-integer linear programming (MILP) approach to automatically synthesize register-transfer level datapaths, given a data flow/control flow graph description of the hardware.

The approach involves developing various timing relationships to be satisfied, but does not include interconnection styles or delays, and does not consider the detailed timing of multiple outputs. Our approach is similar. The differences between their approach and our approach are discussed in Section 3.2.5. Some strategies for improving the computational performance of Hafer's MILP model are reported by Prakash [Pra87]. Hwang et al. [HHL90, HLH91] have described an integer linear programming model for the scheduling problem in data path synthesis under resource constraints and time constraints, and they present a heuristic technique called *Zone Scheduling* for solving large size problems.

Datapath synthesis research has been at the core of the ADAM system [JKMP89, PHJ+88]. In the ADAM system, given a data flow/control flow graph description and the desired constraints on the cost and performance, pipelined datapaths are automatically synthesized by Sehwa [PP86] and non-pipelined by MAHA [PPM86]. Some lower bounds on the cost and performance of the datapaths are reported in [JPP87, JMP88].

The CATHEDRAL-II system [RM87] synthesizes a multiprocessor architecture, however, the architecture is almost completely fixed. It uses a set of synchronous interprocessor communication protocols as opposed to the asynchronous protocols that we propose to use. Its design philosophy is that efficient design synthesis is possible when targeted towards one particular, well defined system architecture. So, it synthesizes customized multiprocessor architectures and each processor is optimized to perform one particular part of the algorithm. The data paths of the processors are tailored to the application by connecting a set of Execution Units (EXU's). The set of available EXU's is restricted to six. It is a rule-based system and the emphasis is on how to optimize each processor rather than on how to configure the overall system. Haroun and Elmasry [HE89] mention multiprocessor architecture synthesis for DSP applications, however, this paper primarily concentrates on the design of an individual processor within

the system, not how to configure the overall system. Selection of CPU design styles was researched by Thomas[Tho77] and implemented by Lawson[Law78].

2.3 Array-Processor Synthesis Research

There has been research effort directed towards synthesis of array-processor architectures. Such architectures are usually characterized by synchronized operation, and all the processors perform nearly identical and relatively simple computations. Synchronous operation makes array-processors best-suited for computations displaying reasonably regular structure and flow of data.

In general, for a given computational task, parallelism could be exploited at different levels. The first level of parallelism is offered by partitioning the task into smaller subtasks. The second level is found within each subtask. The regularity desirable for array-processing is usually found in exploiting the second level of parallelism. At the first level (exploiting parallelism among major subtasks), regularity is less common. It is unusual for a task's major subtasks to be identical or similar. More often a task contains a mix of quite different subtasks with quite different processing requirements and consequently heterogeneous systems are more suitable. Our research is geared towards the synthesis of such heterogeneous systems.

Kung and Leiserson [Kun79, KL80, Kun82] proposed a number of ad hoc special-purpose VLSI systolic array architectures for some important algorithms such as matrix-vector, matrix-matrix multiplications, LU decompositions, recurrence evaluations, and others. Kung et al. [KAGER82] describe a *Wavefront Array Processor* which is a programmable special-purpose multiprocessor array suited for recursive and local data-dependent algorithms. Such algorithms exhibit a continuously advancing wave of data and computational activity, or a *computational wavefront.* This notion of computational wavefront is also discussed in

[WD81]. In [Kun84], Kung proposes a methodology for converting parallel recursive algorithms into synchronous systolic arrays or data-driven wavefront arrays, using the concept of computational wavefront. Johnsson and Cohen [JC81] start with an algorithm-representation and map it onto a VLSI array architecture by using expression manipulation and operator calculus. Moldovan and Fortes [Mol82b, Mol82a, Mol83, For83, FM85] describe a synthesis technique for mapping of cyclic loop algorithms into special-purpose systolic arrays. The technique works by modifying the algorithm using a transformation function which is selected to minimize processing time and interconnection complexity for VLSI arrays. The transformation function is selected to expose hidden parallelism in the algorithm, by using parallelism detection techniques based on algorithm data dependencies. Cappello and Steiglitz [CS83] transform the given algorithm into an abstract model and then apply geometric transforms to design systolic architectures. Miranker and Winkler [MW84] extend Fortes's approach to develop a more generalized theory and methodology for mapping a given algorithm into a systolic array. Li and Wah [LW85] describe a systematic methodology for the design of optimal pure planar systolic arrays for algorithms that are representable as linear recurrence processes. They formulate the design problem as a constrained optimization problem in terms of the systolic array parameters.

2.4 System-Level CAD Tools

There are some system-level CAD tools available. These tools assist the designer during the design process; they do not automate the process.

The configuration of existing components using predesigned interconnection strategies was the subject of the R1 expert system [McD82], a successful package used by DEC to configure the systems it markets. Sara is a well-known system-level tool package which supports the designer in making design decisions, but which makes no design decisions of its own[Est78, EFRV86]. ADAS [Cen88] is a commercial system-level package which supports the designer with

representation and simulation tools. It is a methodology and supporting tools set for system design. It uses directed graph models for design construction and analysis. Essentially, the model used is a Petri net-like model. ADAS does not automate the design activity; it basically provides a CAD environment where the designer iterates through the design process. Thus, the designer constructs and simulates various designs using the ADAS tools until (s)he finds a satisfactory design. The most important tool provided by ADAS is the simulator which is essentially a Petri net simulator.

Chapter 3

The SOS Approach

As outlined in Chapter 1, the approach adopted in this research towards the problem of design of application-specific multiprocessor systems is one of "synthesis." Traditionally (as described in Chapter 2), the research related to multiprocessor systems has concentrated on how to use a given system effectively for a given application. The research described here presents a new viewpoint: *synthesize a system most effective for the given application.*

The approach involves development of a formal model for the multiprocessor synthesis problem. The motivation for a formal model consists of several reasons:

- The design space is large; i.e., the number of possible system designs is large. It is easier to capture the whole design space using a formal model.
- The design must meet constraints; i.e., the designed system must meet arbitrary constraints imposed by the designer. A formal model will be capable of taking such constraints into account in a rigorous way.
- Formal models support deep understanding of the problem (as all the issues/aspects related to the problem which are included in the model are brought out clearly and explicitly in a formal model).
- The understanding offered by the formal model also provides insight as to heuristics which can be applied to solve the problem.

The SOS approach uses mathematical programming to develop the formal model for the synthesis problem. Ideally, one would like to have a completely general mathematical programming model capturing the completely general multiprocessor synthesis problem. However, keeping in mind the practicality of solving such a general and large mathematical programming model, the approach is aimed at developing specialized models for somewhat restricted synthesis problems. Of course, the approach itself is quite general and flexible, and can be applied to varying design situations and design styles. Depending on the specific design problem at hand, a specific model can be created. The challenge is to come up with the specific mathematical programming model applicable to the design problem at hand. The answer lies in the general characteristics of the application tasks as well as the systems being considered. Let us refer to the relevant characteristics of the application tasks as the *task model*, and those of systems as the *system model.* The task model describes the computational model underlying the application task. The system model describes the style of architecture used to implement the task. The task model and the system model together determine the mathematical programming model to be created. It is our assertion that given a task model and a system model, a mathematical programming model can indeed be created capturing the task and the system characteristics along with the design rules to be followed during the system design, and that solution of such a model as a constrained optimization problem will produce an optimal system design.

3.1 Overview of the SOS Approach

SOS assumes the application domain is specified in terms of a task data flow graph (Figure 3.1). The task data flow graph specifies a set of subtasks (nodes in the graph) that need to be performed and the data transfer as well as precedence between them (arcs in the graph). Given the task data flow graph, the

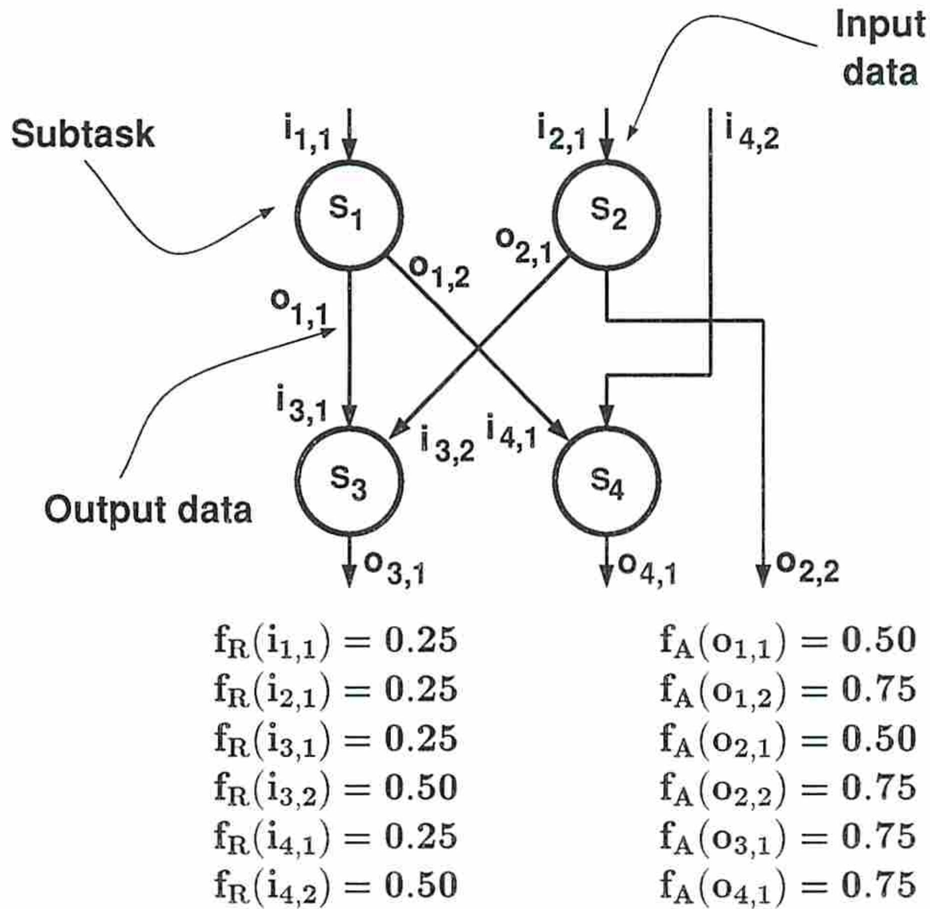
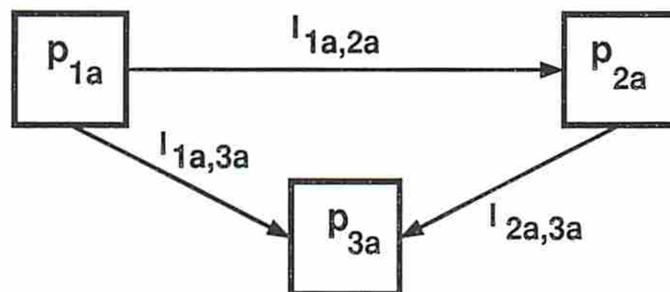


Figure 3.1: Example 1 Task Graph

goal is to synthesize a multiprocessor system which meets various cost and performance constraints. The multiprocessor system is specified in terms of a set of components and the interconnections between them. An example system is shown in Figure 3.2. Synthesizing a system involves making decisions about the number and types of components, the overall interconnection between the components, and the scheduling of subtasks and data transfers on the components.

The SOS approach involves creation of a formal model of the multiprocessor synthesis problem using mathematical programming and the solution of this model. This approach is a natural outgrowth of the work described by Chu



A possible mapping & ordering:

* s_1 performed on p_{1a} ; s_2 , s_4 on p_{2a} in that order; s_3 on p_{3a}
* $i_{4,1}$ transmitted on $l_{1a,2a}$; $i_{3,1}$ on $l_{1a,3a}$; $i_{3,2}$ on $l_{2a,3a}$

Figure 3.2: Synthesized Multiprocessor System I for Example 1 Graph

[CHLE80], Talukdar [MT82], and Hafer [HP83]. Hafer's notation has been adopted wherever meaningful to do so. *This research focuses on the automatic design of the multiprocessor system itself, not merely the mapping of tasks onto a given system.* The SOS approach can be used to explore different interconnection styles; e.g., bus, point-to-point, ring, or a mixture of these. SOS assumes there is no global clock and communications between subtasks are asynchronous at the task level. A distinguishing feature of the research is the fact that SOS designs a truly heterogeneous system, which allows a more precise tailoring of the synthesized system to a specific application.

A complete mathematical programming formulation of the problem requires specification of an objective function that has to be optimized and a set of constraints that have to be satisfied. The objective function can be any function which can be linearized; e.g., the total system cost, or the overall system performance. The set of constraints consists of the correctness constraints that must be satisfied for the overall task to be performed correctly as well as the arbitrary

timing and cost constraints imposed by the designer. The correctness constraints consist primarily of the relations that ensure proper ordering of the subtasks and the data transfers, taking into account the timing involved and the relations that express the conditions for complete and correct system configuration. In order to express the various constraints and the objective function, SOS defines certain variables related to the system. The necessary variables fall into two basic categories: the *timing variables* (real variables which represent timings of various critical events in the operation of the system) and the *binary variables* (0-1 variables which represent the implementation decisions regarding the system configuration). Several constraints comprising the mathematical programming model turn out to be non-linear relations. These relations are linearized and the model is converted into an MILP (Mixed Integer-Linear Programming) formulation. *Bozo*[3.1][HH90] solves the MILP model by invoking a commercial linear programming package, XLP, developed by XMP Software, Inc.

The strength of SOS lies in the fact that it is quite general and flexible. The approach allows us to modify, extend and enhance the model to include more design possibilities and variations easily. The exact form of constraints used can be tailored to meet the characteristics of the design problem at hand. Also, the approach offers a great degree of flexibility in handling arbitrary constraints as they can be expressed using the timing and binary variables defined in the model.

SOS research described in this thesis has been reported extensively in the literature [PP92b, PP92d, PP91a, PP92a, PP92c, PP91b, PP90].

3.2 A Specific SOS Model

In this section, a specific SOS model is presented which assumes point-to-point interconnection; i.e., if a processor p_{d1} needs to send data to another processor p_{d2}, then there must be a direct communication link from p_{d1} to p_{d2}. As

[3.1] A branch-and-bound program to solve MILP problems; developed by L. J. Hafer of Simon Fraser University.

mentioned earlier, the task model and the system model together determine the mathematical programming model; so they are presented first followed by the mathematical programming model. The SOS model presented here has been reported in [PP92b].

3.2.1 A Representative Task Model

The task consists of a set of subtasks. Each subtask requires certain input data and produces certain output data. Inputs to a subtask may come from other subtasks and outputs from a subtask may go to other subtasks. The set of subtasks and the input-output relationships among them can be expressed by a task data flow graph (directed acyclic graph) as shown in Figure 3.1. The subtask nodes are labelled S_1, S_2, etc. (S_a in general). The input end of a data arc is labelled $i_{a,b}$ if it provides the bth input to subtask S_a, and the output end is labelled $o_{a,c}$ if it transmits the cth output from the subtask S_a. Although we represent the task by a data flow graph, we consider a subtle distinction between our model and the traditional data flow model. With the traditional meaning, a subtask would require all the inputs before starting its execution and none of the outputs would be available until after its execution was over. However, in our model subtasks do not require all the inputs before starting their execution and they may produce some outputs even before their completion. To express this possibility, each input $i_{a,b}$ has a parameter $f_R(i_{a,b})$ associated with it which is the fraction of the subtask S_a that can proceed without requiring the input $i_{a,b}$. Similarly, each output $o_{a,c}$ has a parameter $f_A(o_{a,c})$ associated with it which specifies that the output $o_{a,c}$ becomes available when $f_A(o_{a,c})$ fraction of the subtask S_a is completed. For each subtask S_a, a set P_a represents the set of processors capable of executing it. A data arc from node S_{a1} to node S_{a2} implies that some data is transferred from the subtask S_{a1} to the subtask S_{a2}. The volume of data transferred varies from arc to arc, and a parameter $V_{a1,a2}$ specifying the volume is associated with each arc.

3.2.2 A Representative System Model

The multiprocessor system is specified in terms of the processors selected and the interconnection architecture between them. For the specific style under consideration, we assume point-to-point interconnection; i.e., if a processor p_{d1} needs to send data to another processor p_{d2}, then there must be a direct communication link from p_{d1} to p_{d2}. Each processor is assumed to have local memory, and all the interprocessor communication takes place by message-passing over communication links. The subtasks get executed on the processors and the necessary data transfers take place over the communication links.

A processor could be executing at most one subtask at any given time. Also, one and only one processor performs a given subtask. So, once execution of a subtask begins on a processor, the subtask occupies the processor for an uninterrupted duration of time before it completes. The length of the duration is equal to the execution time of the subtask, which depends on the processor type on which it is performed. A parameter, denoted as $D_{PS}(P_t, S_a)$, specifies the execution time for the subtask S_a if processor type P_t is selected to perform it. If two subtasks are to be executed by the same processor, one must be scheduled to begin after the other is completed.

The data transfer corresponding to an arc from subtask S_{a1} to subtask S_{a2} may be a *remote transfer* (if S_{a1} and S_{a2} are mapped to different processors); or it may be a *local transfer* within the same processor (if S_{a1} and S_{a2} are mapped to the same processor). Delay associated with a data transfer depends on whether it is a remote transfer or a local transfer[3.2]. The local transfer delay is represented by the parameter D_{CL} which specifies the time taken in transferring a unit volume of data locally. The remote transfer delay is represented by the parameter D_{CR} which specifies the time taken in transferring a unit volume of data remotely. In practice, the time spent in performing a remote data transfer depends on the amount of traffic in the interconnection network; if two data transfers are

[3.2] Local transfer delay could be negligible compared to the remote transfer delay.

supposed to take place over the same communication link at the same time, then the second can only start after the first is completed (The second set of data will remain held in the local memory of the processor producing it). Essentially, the time spent in remote transfer consists of the *waiting time* and the *actual transfer time*. The parameter D_{CR} only captures the actual transfer time component. The waiting time component is captured in the mathematical programming model as a delay in scheduling the communication by enforcing exclusion in the usage of the communication links. Similar to subtask execution, once a data transfer operation begins, the communication link is released only after the operation is completed; i.e., the link is busy for an uninterrupted duration of $D_{CR}V_{a1,a2}$ time units if $V_{a1,a2}$ is the volume of data associated with the operation.

Our system model assumes overlap between computation and I/O operations. Data transfer operations are taken care of by I/O modules. A subtask can produce output data at intermediate points of its execution. Transfer of such output data can start as soon as it is available (obviously, only if required communication links are available) without delaying the completion of the subtask. It is not necessary to wait for the completion of the subtask before starting the transfer of the output data, since it is assumed the processor executing this subtask does not get involved in the data transfer operation. We assume for this specific model that each processor in the system will have the necessary I/O modules. Similarly, the processor receiving the data does not get involved in the data transfer operation, and could be performing computations while the data is being received by its I/O module (and so the inputs of the subtask could arrive after the processor has already started executing the subtask).

A set P represents the set of all the processors (with varying functionality, cost and performance) available for selection as part of the synthesized system, where $P = \bigcup_a P_a$. Associated with each processor $p_d \in P$ is a parameter C_d which specifies the cost of the processor. C_L specifies the cost of creating a communication link between two processors.

3.2.3 The Mathematical Programming Model

3.2.3.1 The Constraints

In order to express the various constraints, the following variables need to be defined.

Timing Variables: There are three classes of timing variables.

- *Data availability timing variables:*
  - *Input data availability, $T_{IA}(i_{a,b})$:* Time when the data required by input $i_{a,b}$ of subtask S_a is available for use.
  - *Output data availability, $T_{OA}(o_{a,c})$:* Time when the output data value $o_{a,c}$ computed by subtask S_a has become available.
- *Subtask execution timing variables:*
  - *Subtask execution start, $T_{SS}(S_a)$:* Time when the execution of subtask S_a actually begins.
  - *Subtask execution end, $T_{SE}(S_a)$:* Time when the execution of subtask S_a is completed.
- *Data transfer timing variables:*
  - *Data transfer start, $T_{CS}(i_{a,b})$:* Time when the transfer of the data required by input $i_{a,b}$ of subtask S_a actually begins.
  - *Data transfer end, $T_{CE}(i_{a,b})$:* Time when the transfer of the data required by input $i_{a,b}$ of subtask S_a ends.

Figure 3.3 provides a pictorial view of the timing variables and general relationship between them.

Binary Variables: There are two types of binary variables.

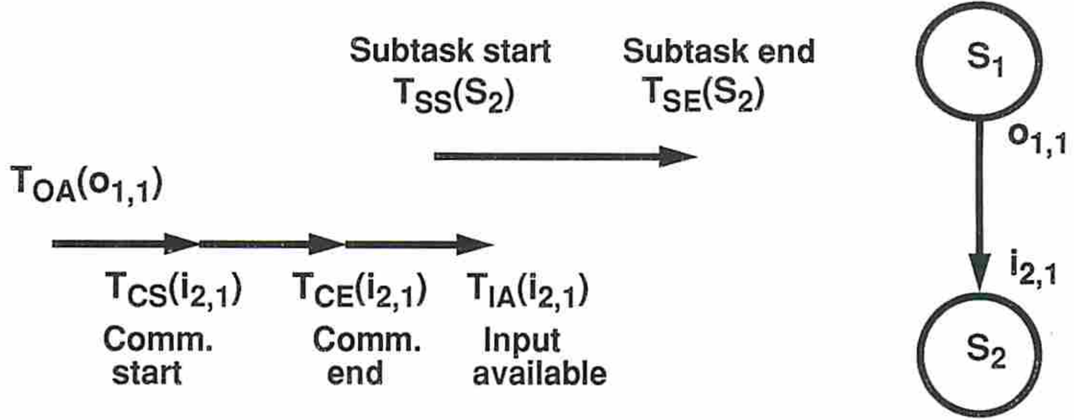


Figure 3.3: Timing Variables

- *Subtask-to-processor-mapping variable, $\sigma_{d,a}$:* The variables of this type specify the mapping between the subtasks and the processors. $\sigma_{d,a} = 1$ indicates processor p_d will implement subtask S_a.

- *Data-transfer-type variable, $\gamma_{a1,a2}$:* The variables of this type specify the data transfer type for the various data arcs. $\gamma_{a1,a2} = 1(0)$ indicates that data transfer from subtask S_{a1} to subtask S_{a2} is a remote (local) transfer.

The necessary **constraints** have been classified into ten categories as follows.

Processor-selection constraint: For each subtask S_a, a set of processors P_a is available to implement it. In order for the implementation to be correct, one and only one processor should be selected to implement the subtask. Thus, for each subtask S_a, the following must be satisfied:

$$\sum_{d|p_d \in P_a} \sigma_{d,a} = 1 \tag{3.1}$$

Figure 3.4 illustrates this constraint by an example.

Data-transfer-type constraint: $\gamma_{a1,a2}$ is a variable which indicates whether the data transfer from the subtask S_{a1} to the subtask S_{a2} is a local transfer or a remote transfer. Now, if the subtasks S_{a1} and S_{a2} are mapped to the same

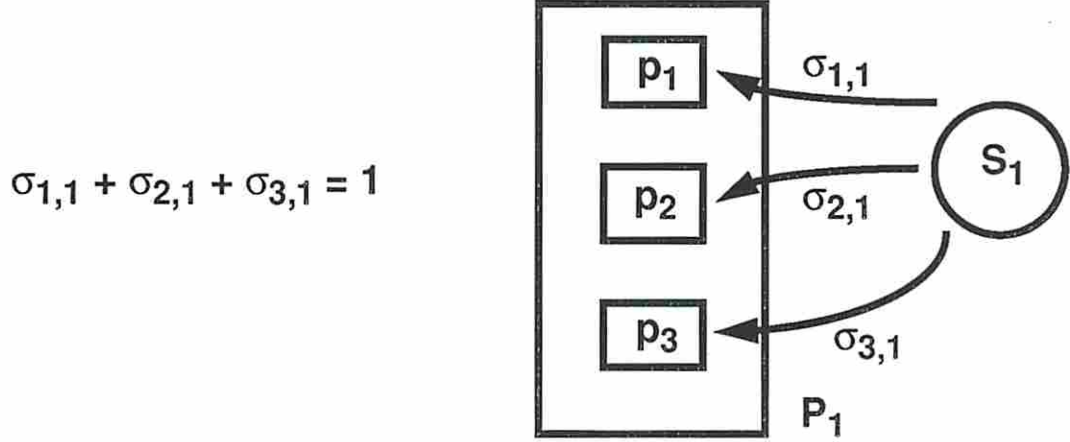


Figure 3.4: Processor-selection Constraint

processor (say p_d, where $p_d \in P_{a1}$ and $p_d \in P_{a2}$), then we know that it is a local transfer, and thus $\gamma_{a1,a2} = 0$. However, if they are mapped to different processors, then the data transfer is remote, and thus $\gamma_{a1,a2} = 1$. Thus, the defining equation for $\gamma_{a1,a2}$ is:

$$\gamma_{a1,a2} = 1 - \sum_{d | p_d \in P_{a1} \cap P_{a2}} \sigma_{d,a1} \sigma_{d,a2} \tag{3.2}$$

We will have such an equation for each pair of subtasks communicating with each other.

Input-availability constraint: $T_{IA}(i_{a,b})$ is the time the data required at input $i_{a,b}$ will be available, which will be the time $T_{CE}(i_{a,b})$ when the data transfer has ended. So, for each input $i_{a,b}$, we have:

$$T_{IA}(i_{a,b}) = T_{CE}(i_{a,b}) \tag{3.3}$$

Output-availability constraint: Once execution of the subtask S_a begins, a certain time elapses before an output data value $o_{a,c}$ produced by the subtask becomes available. The time elapsed would be the time taken in executing $f_A(o_{a,c})$

$$T_{OA}(o_{a,c}) = T_{SS}(S_a) + f_A(o_{a,c})\ (T_{SE}(S_a) - T_{SS}(S_a))$$

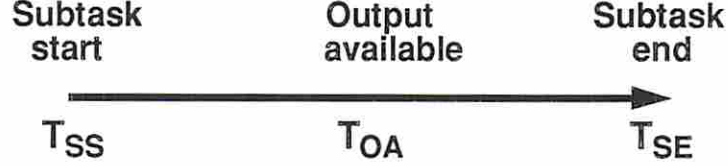


Figure 3.5: Output-availability Constraint

fraction of the subtask; and so the time $T_{OA}(o_{a,c})$ must satisfy the following relation (we will have such a relation for each output):

$$T_{OA}(o_{a,c}) = T_{SS}(S_a) + f_A(o_{a,c})(T_{SE}(S_a) - T_{SS}(S_a)) \tag{3.4}$$

Figure 3.5 illustrates the general relationship involved pictorially.

Subtask-execution-start constraint: $T_{SS}(S_a)$ is the time the subtask S_a begins execution. There must be a certain relationship between the time a given subtask begins its execution and the times at which its various inputs become available. Since $f_R(i_{a,b})$ fraction of the subtask S_a can proceed without requiring the input $i_{a,b}$, the following relation must be satisfied for all the inputs $i_{a,b}$ to the subtask:

$$T_{IA}(i_{a,b}) \leq T_{SS}(S_a) + f_R(i_{a,b})(T_{SE}(S_a) - T_{SS}(S_a)) \tag{3.5}$$

Figure 3.6 illustrates the general relationship involved pictorially.

Subtask-execution-end constraint: Once execution of a subtask begins, a time equal to the execution time of the subtask must elapse before the subtask is completed. Execution time of the subtask depends on the processor type being used for it. *A priori* we do not know which processor type a given subtask S_a is going to be mapped to. Any processor from the set P_a could be selected

Figure 3.6: Subtask-execution-start Constraint

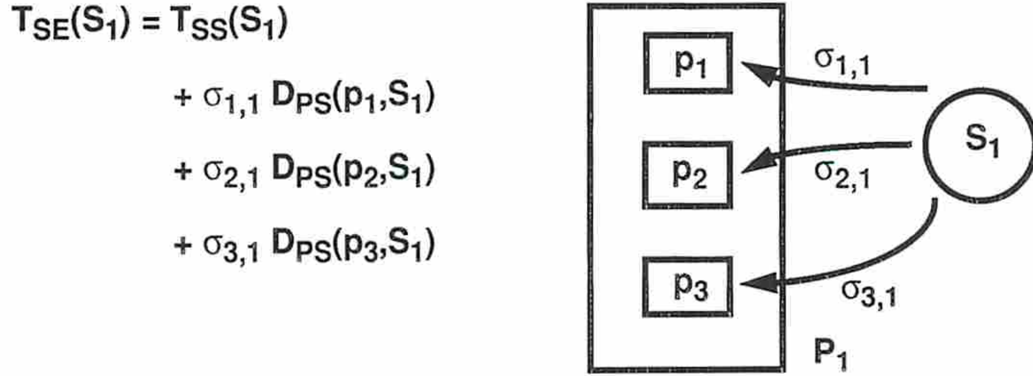


Figure 3.7: Subtask-execution-end Constraint

to execute the subtask S_a. The uncertainty can be expressed by the following relation (where $Typ(p_d)$ represents the type of the processor p_d). The summation acts as a selection since only one $\sigma_{d,a} = 1$ for each a (for each subtask S_a, we need such a relation):

$$T_{SE}(S_a) = T_{SS}(S_a) + \sum_{d|p_d \in P_a} \sigma_{d,a} D_{PS}(Typ(p_d), S_a) \tag{3.6}$$

Figure 3.7 illustrates this constraint by an example.

- $T_{CS}(i_{3,1})$: **Time when the communication/transfer of the data required by input $i_{3,1}$ of subtask S_3 actually begins**
- $T_{OA}(o_{1,1})$: **Time when the output data value $o_{1,1}$ computed by S_1 has become available**

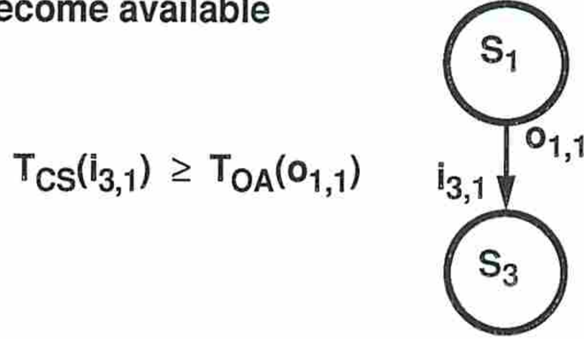


Figure 3.8: Data-transfer-start Constraint

Data-transfer-start constraint: The time at which transfer of data begins must be after the output data is produced. Except for external inputs, for each input data $i_{a2,b2}$ (to the subtask S_{a2}) being supplied by another subtask's output, if the output supplying the data is $o_{a1,c1}$, the following relation must be satisfied by $T_{CS}(i_{a2,b2})$, the start of the data transfer:

$$T_{CS}(i_{a2,b2}) \geq T_{OA}(o_{a1,c1}) \tag{3.7}$$

Figure 3.8 illustrates this constraint by an example.

Data-transfer-end constraint: The time at which transfer of data ends, T_{CE}, depends on whether the transfer is remote or local. *A priori* we do not know which case will occur. However, the two possibilities can be combined into one single relation using the variable $\gamma_{a1,a2}$. Thus, for each input data $i_{a2,b2}$ being supplied by another subtask S_{a1}, we have:

$$T_{CE}(i_{a2,b2}) = T_{CS}(i_{a2,b2}) + \gamma_{a1,a2} D_{CR} V_{a1,a2} + (1 - \gamma_{a1,a2}) D_{CL} V_{a1,a2} \tag{3.8}$$

$$T_{CE}(i_{3,2}) = T_{CS}(i_{3,2}) + \Upsilon_{2,3} D_{CR} V_{2,1} + (1 - \Upsilon_{2,3}) D_{CL} V_{2,1}$$

T_{CE}: Time at which data transfer is completed

T_{CS}: Time at which data transfer starts

$V_{2,1}$: Volume of data at the output $o_{2,1}$

D_{CR}: Time spent in transferring a unit volume of data remotely

D_{CL}: Time spent in transferring a unit volume of data locally

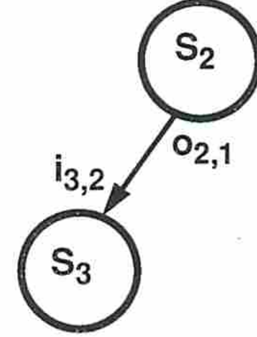


Figure 3.9: Data-transfer-end Constraint

Figure 3.9 illustrates this constraint by an example.

The next two categories of constraints ensure that the hardware resources (processors, communication links) are shared correctly. These constraints ensure that the same hardware resource is not scheduled to perform more than one function during any given time interval. In order to express these constraints concisely, we define a special function called an overlap function L (as defined in [HP83]). The function is defined on two closed intervals of time, $[t1, t2]$ and $[t3, t4]$ (where $t1 < t2$ and $t3 < t4$), as:

$$L([t1, t2], [t3, t4]) = \begin{cases} 1, & \text{if the intervals overlap} \\ 0, & \text{otherwise} \end{cases}$$

Processor-usage-exclusion constraint: If two subtasks S_{a1} and S_{a2} are being executed by the same processor p_d, then the two subtasks must not be scheduled to be executed at the same time. The situation that two subtasks S_{a1} and S_{a2} are being implemented by the same processor p_d implies $\sigma_{d,a1} = \sigma_{d,a2} = 1$. For each processor p_d and each pair of subtasks S_{a1} and S_{a2} such that the sets of

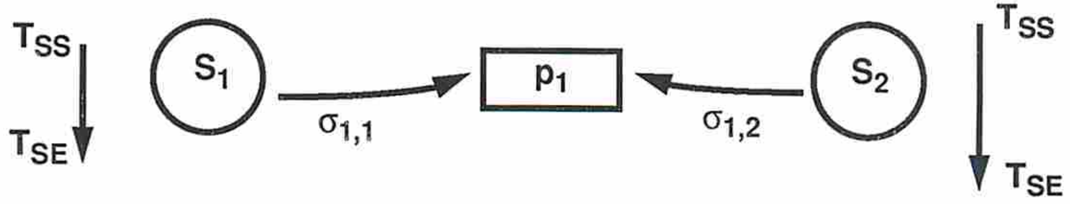


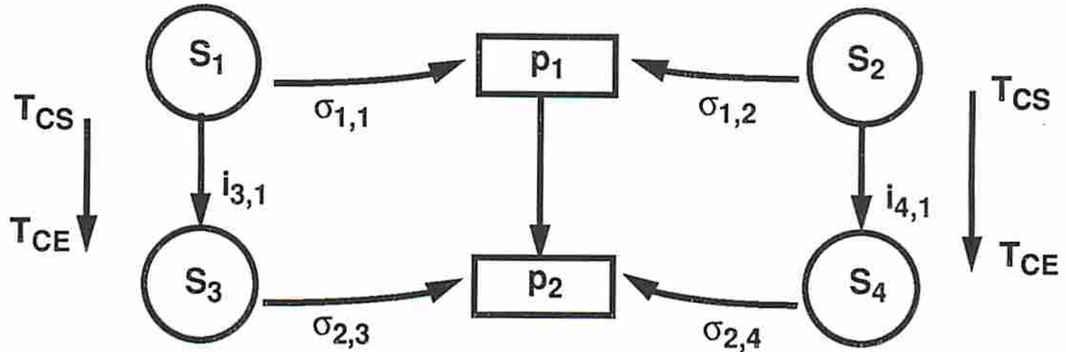
Figure 3.10: Processor-usage-exclusion Constraint

processors P_{a1} and P_{a2} available to implement the subtasks contain the processor p_d, the following relation ensures that the overlap in the usage of the processor by the two subtasks is prevented:

$$\sigma_{d,a1}\sigma_{d,a2}L([T_{SS}(S_{a1}),T_{SE}(S_{a1})],[T_{SS}(S_{a2}),T_{SE}(S_{a2})]) = 0 \tag{3.9}$$

Figure 3.10 illustrates this constraint by an example.

Communication-link-usage-exclusion constraint: If the data required by two inputs $i_{a1,b1}$ and $i_{a2,b2}$ are being transmitted over the same communication link, then the two data transfers must not be scheduled at the same time. Let us say the input data $i_{a1,b1}$ is supplied by the subtask S_{a3} and the input data $i_{a2,b2}$ is supplied by the subtask S_{a4}. The two inputs $i_{a1,b1}$ and $i_{a2,b2}$ will be transmitted over the same communication link if the two subtasks S_{a1} and S_{a2} are mapped to the same processor, say p_{d2}, and also the subtasks S_{a3} and S_{a4} are mapped to the same processor, say p_{d1} (in that case, both the inputs will be transmitted over the communication link from processor p_{d1} to processor p_{d2}). So, for each processor pair (p_{d1}, p_{d2}) and each pair of inputs $i_{a1,b1}$ and $i_{a2,b2}$, if the input $i_{a1,b1}$ is being supplied from S_{a3} to S_{a1} and the input $i_{a2,b2}$ from S_{a4} to S_{a2} then the following relation ensures that the overlap in the usage of the communication link from processor p_{d1} to processor p_{d2} by the two data transfers is prevented:



$$\sigma_{1,1}\,\sigma_{1,2}\,\sigma_{2,3}\,\sigma_{2,4}\,L([T_{CS}(i_{3,1}),T_{CE}(i_{3,1})],[T_{CS}(i_{4,1}),T_{CE}(i_{4,1})]) = 0$$

Figure 3.11: Communication-link-usage-exclusion Constraint

$$\sigma_{d2,a1}\sigma_{d2,a2}\sigma_{d1,a3}\sigma_{d1,a4}L([T_{CS}(i_{a1,b1}),T_{CE}(i_{a1,b1})],[T_{CS}(i_{a2,b2}),T_{CE}(i_{a2,b2})]) = 0 \tag{3.10}$$

The above constraint also captures the waiting times associated with the remote data transfers. Figure 3.11 illustrates this constraint by an example.

3.2.3.2 Objective Functions

Two of the most important goals that the designer may wish to optimize are the overall system performance and the total system cost.

Overall System Performance: The performance is frequently measured by how fast the system can perform the given task, the time at which the task is completed (or all the subtasks are completed). If T_F is a real variable representing the time at which the task is completed, then the objective is to *minimize* T_F. To ensure that T_F represents the time at which all the subtasks are completed, we need to introduce the following constraint in the model (for each subtask S_a):

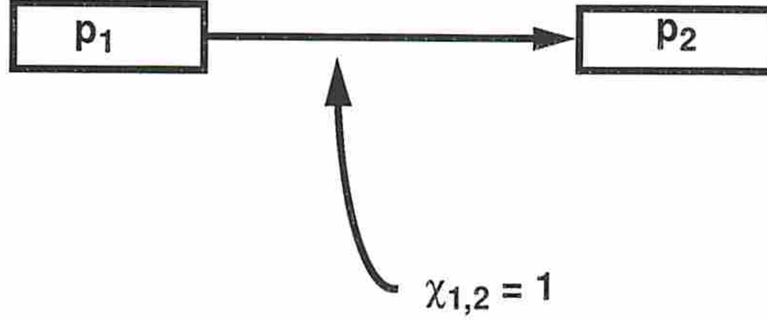


Figure 3.12: Communication-link-creation variable

$$T_F \geq T_{SE}(S_a) \tag{3.11}$$

Total System Cost: The total cost of the system can be expressed as the sum of the costs of the processors selected and the costs of the links created. In order to do so, we need to define two types of binary variables as follows.

Processor-selection variable, β_d: The variables of this type specify which processors have been selected in the synthesized architecture. $\beta_d = 1$ indicates the processor p_d is included in the system.

Communication-link-creation variable, $\chi_{d1,d2}$: The variables of this type specify what communication links are present in the synthesized architecture. $\chi_{d1,d2} = 1$ indicates there exists a communication link from the processor p_{d1} to the processor p_{d2} in the designed system. Figure 3.12 illustrates the semantics of this variable.

Using the variables defined above, the objective is to

$$MINIMIZE \sum_{d|p_d \in P} \beta_d C_d + \sum_{d1,d2|p_{d1} \in P \wedge p_{d2} \in P} \chi_{d1,d2} C_L$$

where C_d is the cost of a processor p_d and C_L is the cost of building a link between two processors, as defined in Section 3.2.2. The variables of type β_d

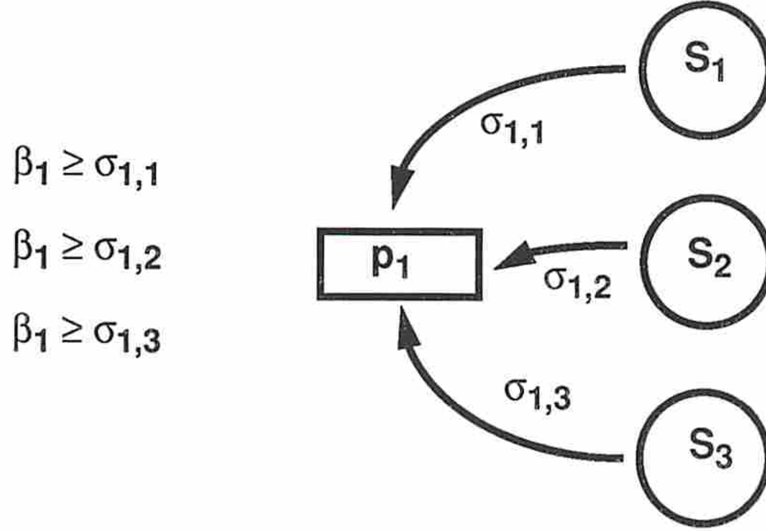


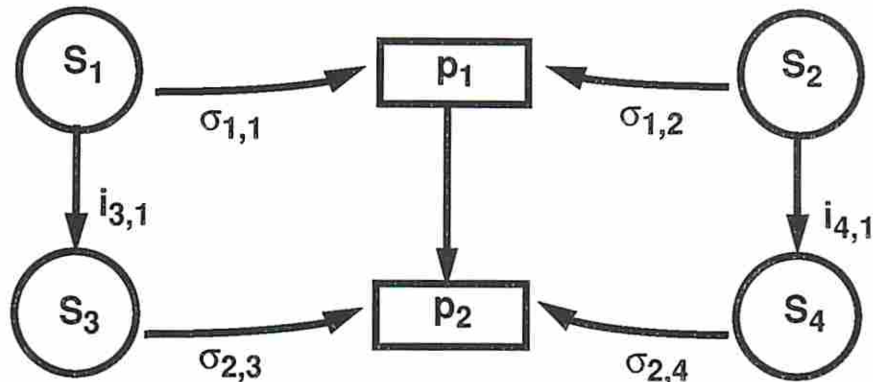
Figure 3.13: Relationship between β-type and σ-type Variables

are related to the variables of type $\sigma_{d,a}$. A processor p_d will be included in the system if and only if at least one of the subtasks S_a ($p_d \in P_a$) is mapped to it, which implies that the variable β_d is the logical OR of all the $\sigma_{d,a}$ variables. This can be expressed by introducing the following constraint in the model (for all a such that $p_d \in P_a$):

$$\beta_d \geq \sigma_{d,a} \tag{3.12}$$

The idea behind this constraint is further illustrated by example in Figure 3.13.

The variables of type $\chi_{d1,d2}$ are also related to the variables of type $\sigma_{d,a}$. A communication link is created from processor p_{d1} to processor p_{d2} if and only if at least one of the subtasks S_{a1} ($p_{d1} \in P_{a1}$) mapped to the processor p_{d1} needs to send data to at least one of the subtasks S_{a2} ($p_{d2} \in P_{a2}$) mapped to the processor p_{d2}. So, the variable $\chi_{d1,d2}$ is the logical OR of all the product terms of the form $(\sigma_{d1,a1}\sigma_{d2,a2})$, where the subtask S_{a1} supplies some data to the subtask S_{a2}. This condition leads to the introduction of following constraint in the model (for all



$\chi_{1,2} \geq \sigma_{1,1}\,\sigma_{2,3}$

$\chi_{1,2} \geq \sigma_{1,2}\,\sigma_{2,4}$

Figure 3.14: Relationship between χ-type and σ-type Variables

$a1, a2$ such that $p_{d1} \in P_{a1}$ and $p_{d2} \in P_{a2}$ and subtask S_{a1} sends data to subtask S_{a2}):

$$\chi_{d1,d2} \geq \sigma_{d1,a1}\sigma_{d2,a2} \tag{3.13}$$

The idea behind this constraint is further illustrated by example in Figure 3.14.

The essence of the model has been presented. It is easy to see that *arbitrary constraints imposed by the designer (within the semantics of the model) can be expressed using the timing and binary variables defined in the model.*

3.2.4 Synthesis Using the Model

3.2.4.1 Linearization of the Model

Several constraints comprising the mathematical programming model presented in Section 3.2.3 are non-linear relations. In order to solve the model as an

MILP, these relations must be linearized and the model converted into an MILP formulation.

Eq. (3.2) is non-linear. It can be linearized by defining a binary variable of the form $\delta_{d,a1,a2}$ for each product of the form $(\sigma_{d,a1}\sigma_{d,a2})$. Using the new variables defined, Eq. (3.2) can be replaced by the following set of linear relations:

$$\gamma_{a1,a2} = 1 - \sum_{d|p_d \in P_{a1} \cap P_{a2}} \delta_{d,a1,a2} \tag{3.14}$$

$$\delta_{d,a1,a2} \leq \sigma_{d,a1} \tag{3.15}$$

$$\delta_{d,a1,a2} \leq \sigma_{d,a2} \tag{3.16}$$

Now, let us consider linearization of Eq. (3.9). The constraint says that if two subtasks S_{a1} and S_{a2} are executed on the same processor p_d, then there must be no overlap in their execution time intervals. This implies that either the start time of S_{a1} is sometime after the completion time of S_{a2} or the start time of S_{a2} is sometime after the completion time of S_{a1}. Let us define a binary variable $\alpha_{a1,a2}$ whose value decides which of the two possibilities occurs. $\alpha_{a1,a2} = 1$ implies S_{a1} is executed first. Using this variable, Eq. (3.9) can be rewritten as the following pair of non-linear relations:

$$\begin{aligned} T_{SS}(S_{a2}) &\geq \alpha_{a1,a2}\sigma_{d,a1}\sigma_{d,a2}T_{SE}(S_{a1}) \\ T_{SS}(S_{a1}) &\geq (1 - \alpha_{a1,a2})\sigma_{d,a1}\sigma_{d,a2}T_{SE}(S_{a2}) \end{aligned}$$

To linearize the above pair, let us define a constant T_M whose value is larger than the possible values of all the timing variables in the model. Now, the following two linear relations express the desired constraint:

$$T_{SS}(S_{a2}) \geq T_{SE}(S_{a1}) - (3 - \alpha_{a1,a2} - \sigma_{d,a1} - \sigma_{d,a2})T_M \tag{3.17}$$

$$T_{SS}(S_{a1}) \geq T_{SE}(S_{a2}) - (2 + \alpha_{a1,a2} - \sigma_{d,a1} - \sigma_{d,a2})T_M \tag{3.18}$$

Similarly, to linearize Eq. (3.10), we need to define a binary variable $\phi_{a1,b1,a2,b2}$. $\phi_{a1,b1,a2,b2} = 1$ indicates the data transfer for $i_{a1,b1}$ takes place before the data transfer for $i_{a2,b2}$ if the same communication link is used for both the transfers. Using this variable, Eq. (3.10) can be rewritten as the following pair of linear relations:

$$\begin{aligned} T_{CS}(i_{a2,b2}) \geq\ & T_{CE}(i_{a1,b1}) - (5 - \phi_{a1,b1,a2,b2} \\ & -\sigma_{d2,a1} - \sigma_{d2,a2} - \sigma_{d1,a3} - \sigma_{d1,a4})T_M \quad (3.19) \\ T_{CS}(i_{a1,b1}) \geq\ & T_{CE}(i_{a2,b2}) - (4 + \phi_{a1,b1,a2,b2} \\ & -\sigma_{d2,a1} - \sigma_{d2,a2} - \sigma_{d1,a3} - \sigma_{d1,a4})T_M \quad (3.20) \end{aligned}$$

Finally, non-linear Eq. (3.13) can be simply rewritten in the following linear form:

$$\chi_{d1,d2} \geq \sigma_{d1,a1} + \sigma_{d2,a2} - 1 \tag{3.21}$$

3.2.4.2 Size of the Linearized MILP Model

Let n be the number of nodes (subtasks) and m the number of edges (data transfers) in the task graph. Let $p = |P|$ be the total number of processors available for selection as part of the system. Then the following statements can be made about the worst case growth of the MILP model.

The number of real (timing) variables grows as $O(an + bm)$, where a and b are some constants. The number of binary variables grows as $O(cn^2 + dm^2 + ep^2)$, where c, d and e are some constants. The number of constraints grows as $O(fp^2m^2 + gpn^2)$, where f and g are some constants. Although these are the

growth rates in the worst case, the actual numbers of variables and constraints are usually fewer (as we will see in the examples presented later).

3.2.4.3 Solution of the Model

The linearized MILP model is solved using the *Bozo* program [HH90]. As a result of solving the model, we get the following information as outputs:

- a multiprocessor system; i.e., the chosen set of processors and the interconnection architecture,
- a schedule for the subtasks, and
- detailed timing information for computation and transfer of data.

Several example systems have been synthesized using the model described here; also some tradeoff studies have been performed. These results are reported in Chapter 4.

3.2.5 Comparison with Hafer's RT-Level Model

As we have mentioned earlier, Hafer and Parker [HP83, Haf81] have used a mathematical programming approach for register-transfer level synthesis. Hafer's RT-level model also involves expressing timing relationships that must be satisfied for a correct and complete design. At first sight, it may seem that our model is a replica of Hafer's model. However, it must be emphasized that such is not the case. There are some fundamental issues that have to be addressed differently at the two levels (RT-level vs. system-level). Some of the differences are listed here:

- Hafer's model does not take into account the delay associated with the interconnection of the hardware elements explicitly. Output of a hardware element (operator or register) is considered to be immediately available at

the input of the hardware element using it. This approach was appropriate at the RT-level as the interconnection delay (wiring and multiplexor delays) is probably much smaller than the delays associated with the hardware elements. However, at the system level we can not ignore the interconnection (communication) delays. Output generated by a processor can not be assumed to be immediately available to another processor; it has to be communicated through the interconnection network and the delay is by no means negligible. So, communication delays have to be explicitly taken into account by the model. Our model makes an attempt in this direction.

- In Hafer's model, an operation can not start until all its inputs have become available. In our model, a subtask can begin its execution even with some (or no) inputs available. At the RT-level, the requirement that all inputs be available before an operation starts is probably acceptable; however, at the system level it is not acceptable as a subtask may not require a particular input until it reaches a certain point in its execution. Theoretically speaking, it is conceivable that even at RT-level there are operations which can start without requiring all inputs. However, the number of such operations and the performance loss seem to be low enough to ignore such operations in order to keep the model simple at the RT-level.

- In Hafer's model, all the outputs of an operation become available only after the operation is completed. In our model, some of the outputs of a subtask can become available even before the subtask is completed. Once again, discussion similar to the previous point applies here.

Chapter 4

Experiments with Point-to-Point Interconnection Model

In this chapter, the results of a set of experiments and tradeoff studies which were performed to validate the SOS approach are presented. The specific SOS model described in Section 3.2 was used in these experiments. Two example task graphs have been used. The first example consists of four subtask nodes, while the second consists of nine. Some of the data related to these examples is taken from [MT82].

4.1 Example 1: Four-Subtask Task Graph

This example data flow graph is shown in Figure 3.1. Associated f_R and f_A parameters are also given in the figure, constraining input/output timing for the subtasks.

We assume we have available three types of processors: p_1, p_2, and p_3. The costs of these processors and the execution times of various subtasks on the processors are given in Table 4.1. An entry of '–' in the table implies that the particular processor is functionally not capable of performing the particular subtask. As is obvious from the table, different processors have different cost-speed-functionality characteristics.

		Execution Time			
Proc.	Cost	S_1	S_2	S_3	S_4
p_1	4	1	1	-	3
p_2	5	3	1	2	1
p_3	2	-	3	1	-

Table 4.1: Processor Characteristics - Example 1

In this example the volume of data that needs to be communicated is one unit at each arc in the graph. Local transfer delay is given to be negligible; i.e., $D_{CL} = 0$. We are also given the communication link characteristics. The cost of a link, C_L, is one unit; and the remote transfer delay for a unit volume of data over a link, D_{CR}, is also one unit.

The MILP model for the example consists of 21 timing and 72 binary variables, and 174 constraints. Bozo was used to generate 4 non-inferior[4.1] systems. These different systems were generated by changing the constraint value for the total cost of the system, and optimizing the overall performance of the system. Bozo's runtime to generate each of these designs is on the order of a few seconds. These runtimes are on a Solbourne Series5e/900 (similar to Sun SPARCsystem 4/490) with 128 MB of memory. Cost, performance and runtime for the four designs are given in Table 4.2. Note that the performance of the target system is measured by the time at which the overall task is completed.

A brief discussion of these designs follows:

- *Design 1:* This design consists of 3 processors: p_{1a} - a processor of type p_1, p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtask S_1, processor p_{2a} performs subtasks S_2 and S_4 in that order, and processor p_{3a} performs subtask S_3. There are three communication links: $l_{1a,2a}$, $l_{1a,3a}$, and $l_{2a,3a}$. Data $i_{4,1}$ gets transmitted on link $l_{1a,2a}$,

[4.1]A system (characterized by its cost and performance) is considered non-inferior if cost (performance) can not be improved without degrading performance (cost).

Design	Runtime (*sec*)	Cost	Performance (time units)
1	11.2	14	2.5
2	24.5	13	3
3	26.5	7	4
4	38.7	5	7

Table 4.2: Example 1 Systems

data $i_{3,1}$ gets transmitted on link $l_{1a,3a}$, and data $i_{3,2}$ gets transmitted on link $l_{2a,3a}$. As an illustration, this system is shown in Figure 4.1. A schedule for the various events is also shown in the figure. Figure 4.1 shows the schedule in a very detailed fashion; all the timing variables and their values are indicated in the figure. The purpose of this is to show how solving the MILP model leads to assignment of values to all the timing variables and hence a detailed schedule. The same schedule is also shown in Figure 4.2 in a more conceptual way; i.e., it only shows what component is occupied performing what event (instead of cluttering the figure with detailed notation and timing variables). The schedule shown clearly indicates the kinds of detailed schedules that can be generated using the SOS approach and the complexity of such schedules. This is another unique feature of SOS.

- *Design 2:* This design is similar to design 1, and also consists of 3 processors: p_{1a}, p_{2a}, and p_{3a}. However, it has only two links: $l_{1a,2a}$, and $l_{1a,3a}$. Presence of fewer links forces a change in the mapping between the resources and the events. Processor p_{1a} performs subtasks S_1 and S_2 in that order, processor p_{2a} performs subtask S_4, and processor p_{3a} performs subtask S_3. Data $i_{4,1}$ gets transmitted on link $l_{1a,2a}$, data $i_{3,1}$ and data $i_{3,2}$ get transmitted on link $l_{1a,3a}$ in that order.

- *Design 3:* This design consists of 2 processors: p_{1a} - a processor of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_1 and S_4 in that order, and processor p_{3a} performs subtasks S_2 and S_3 in that

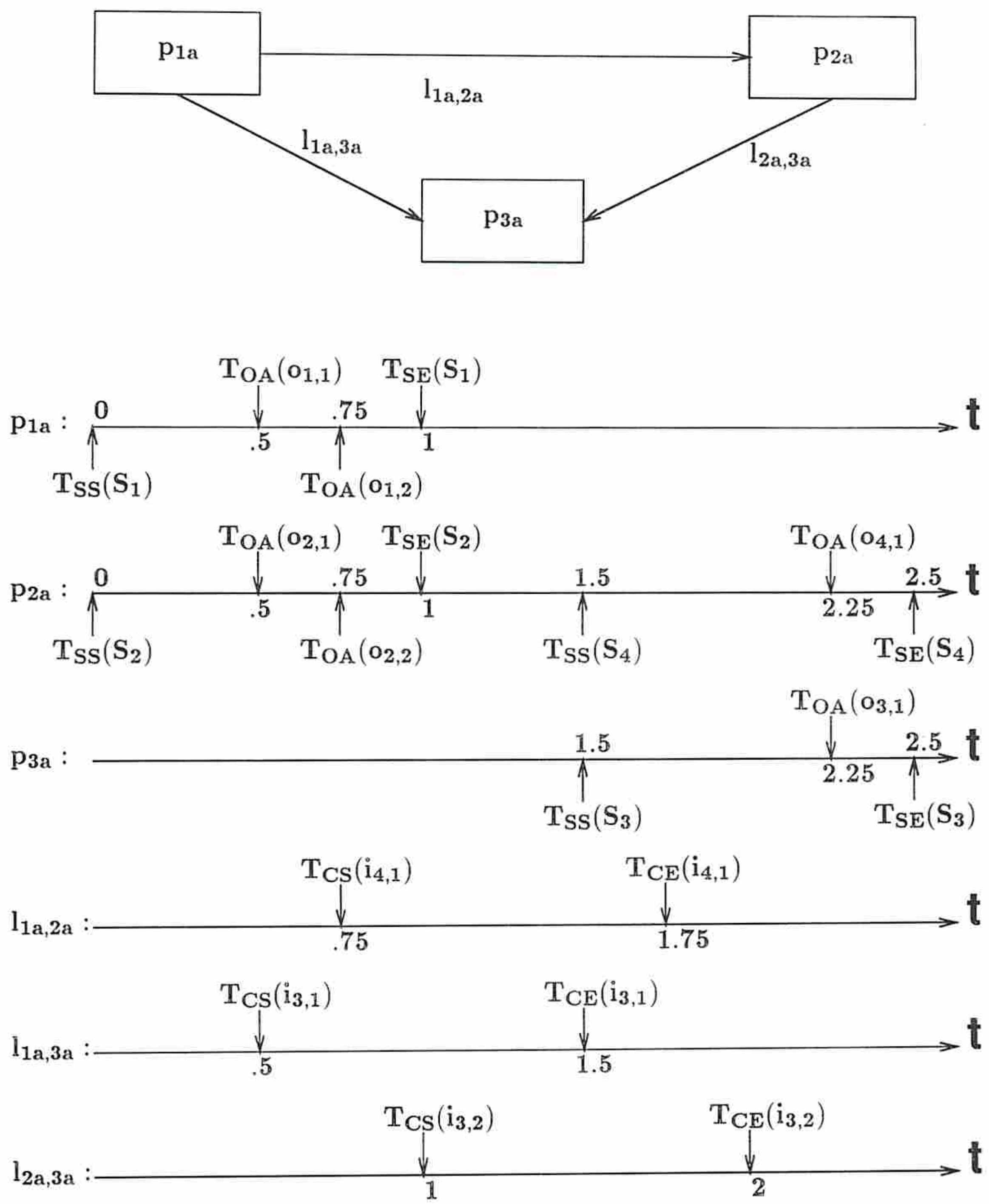


Figure 4.1: Synthesized Multiprocessor System I and Detailed Schedule for Example 1

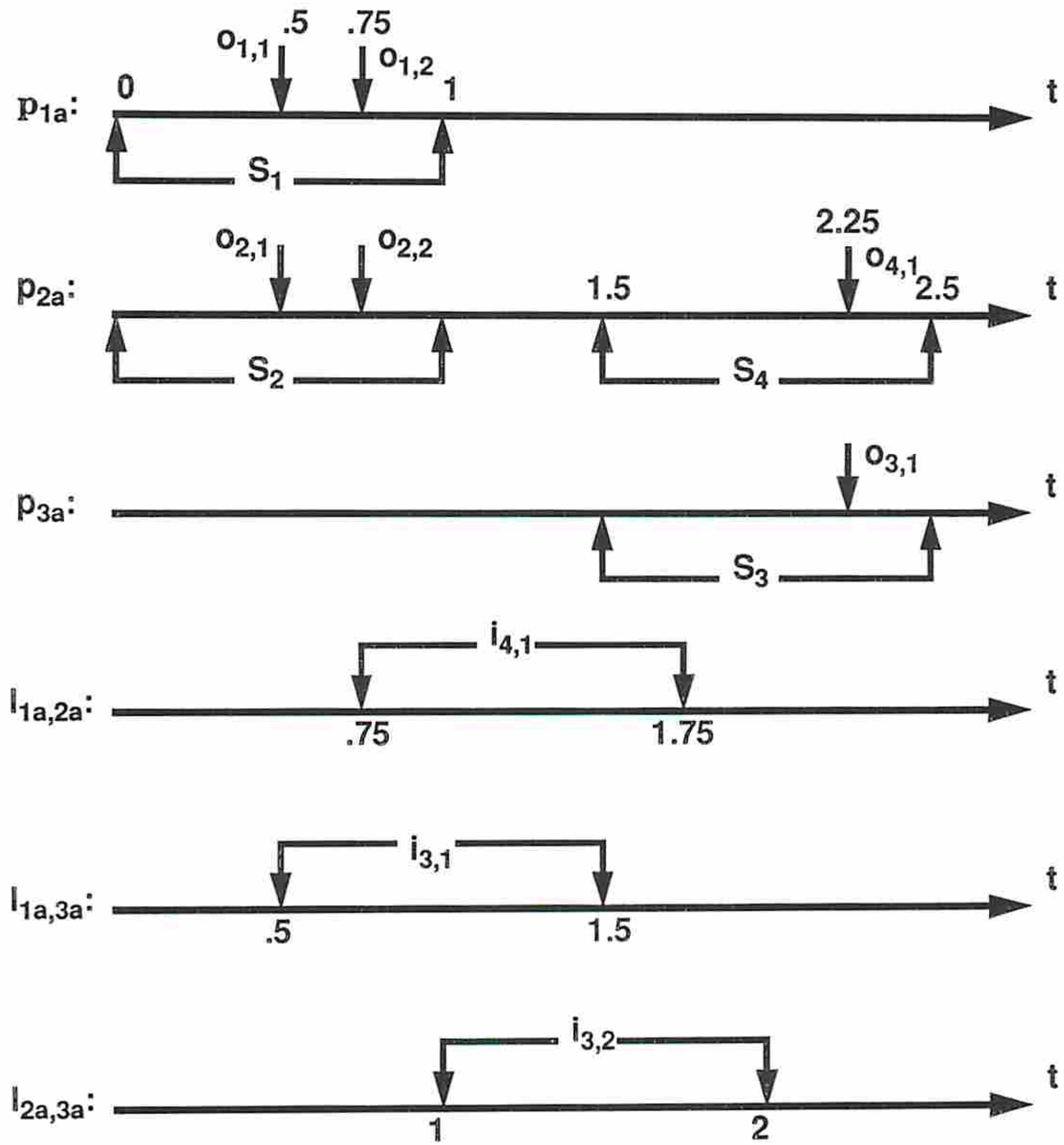


Figure 4.2: System I Schedule (Concise) for Example 1

order. There is a communication link: $l_{1a,3a}$. Data $i_{3,1}$ gets transmitted on link $l_{1a,3a}$.

- *Design 4:* This design consists of just 1 processor: p_{2a} - a processor of type p_2. The processor performs the subtasks S_2, S_1, S_3, and S_4 in that order.

4.2 Tradeoff Studies Using the Four-Subtask Graph

Some tradeoff studies were performed using the example in Section 4.1. We studied the role of inter-subtask communication in synthesis of the systems. The study was performed by varying the ratio between communication times and execution times.

4.2.1 Experiment 1: Increase the Communication Time

In this experiment, we increased the volume of data to be transferred for each arc in the four-subtask task graph. All the other parameters remained as given in Section 4.1.

When the volume of data is doubled for each of the arcs (i.e., the volume is two units instead of one), 3-processor designs become inferior. Only two designs remain non-inferior: the 2-processor design and the uniprocessor design.

The 2-processor design also becomes inferior when the volume of data is made six times the original volume (i.e., the volume is six units for each arc). Only the uniprocessor design remains non-inferior.

4.2.2 Experiment 2: Increase the Execution Time

In this experiment, we increased the size of each of the subtasks (and thus the execution times in Table 4.1). All the other parameters remained as given in Section 4.1.

When the size of each of the subtasks is doubled (and hence the execution time is doubled for each of the processor types), the number of non-inferior designs becomes five (instead of four). The new non-inferior design is a 3-processor design. This design consists of 3 processors: p_{1a}, p_{1b} - two processors of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_1 and S_2 in that order, processor p_{1b} performs subtask S_4, and processor p_{3a} performs subtask S_3. There are two communication links: $l_{1a,1b}$, and $l_{1a,3a}$. Data $i_{4,1}$ gets transmitted on link $l_{1a,1b}$, and data $i_{3,2}$ and data $i_{3,1}$ get transmitted on link $l_{1a,3a}$ in that order.

When the size of each of the subtasks is further increased and made three times the original size, the number of non-inferior designs becomes seven (as opposed to five for the double-size case above). The two new additions are a 4-processor design and a new 2-processor design. The 4-processor design consists of p_{1a}, p_{1b} - two processors of type p_1, p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtask S_1, processor p_{1b} performs subtask S_2, processor p_{2a} performs subtask S_4, and processor p_{3a} performs subtask S_3. There are three communication links: $l_{1a,2a}$, $l_{1a,3a}$, and $l_{1b,3a}$. Data $i_{4,1}$ gets transmitted on link $l_{1a,2a}$, data $i_{3,1}$ gets transmitted on link $l_{1a,3a}$, and data $i_{3,2}$ gets transmitted on link $l_{1b,3a}$. The new 2-processor design consists of p_{1a} - a processor of type p_1, and p_{2a} - a processor of type p_2. Processor p_{1a} performs subtask S_1 and S_2 in that order, and processor p_{2a} performs subtasks S_3 and S_4 in that order. There is a communication link: $l_{1a,2a}$. Data $i_{3,1}$, data $i_{3,2}$ and data $i_{4,1}$ get transmitted on link $l_{1a,2a}$ in that order.

The results of experiments 1 and 2 essentially indicate what is known intuitively, that as the ratios between the inter-subtask communication (in time units) and the sizes of subtasks(in time units) increase, designs with fewer processors are synthesized as they can achieve the desired performance with lower costs. However, as the sizes of subtasks increase (and thus inter-subtask communication becomes relatively less important), multiprocessing becomes useful

and designs with more processors are synthesized as they can provide better performance.

4.3 Some More Tradeoff Studies

In this section, we report some more tradeoff studies performed using the example in Section 4.1. We studied role of various factors in synthesis of the systems. Here, we made an assumption that a subtask requires all the inputs before it can start and that none of the outputs from a subtask become available until its execution is over; i.e, all the f_R and f_A parameters were set to 0 and 1 respectively.

First, we generated all the non-inferior designs under the new assumption. Only 3 non-inferior designs were found instead of 4 (as in Section 4.1). One 3-processor design (design 2 in Section 4.1) becomes inferior. This is intuitively explainable. Setting of f_R and f_A parameters to 0 and 1 respectively essentially amounts to a reduction in the degree of inherent parallelism available in the task. Multiprocessing becomes less useful as there is less parallelism available for exploitation; and hence one 3-processor design becomes inferior. Next we performed some more tradeoff studies by varying values of some parameters.

4.3.1 Variation in Processor Costs

Here, we changed the costs of the processors. Cost parameters for type p_1, p_2, and p_3 are 6, 10, and 2 units respectively.

Now, only 2 non-inferior designs were found instead of 3 described above. The uniprocessor design becomes inferior. This is so because the uniprocessor design (a type p_2 processor) costs much more now than the 2-processor design, and obviously provides worse performance.

4.3.2 Variation in Communication Link Cost/Performance

Here, we changed the communication link characteristics. The link's performance as well as cost was doubled; i.e., $D_{CR} = 0.5$, and $C_L = 2$.

In this case, we found 4 non-inferior designs again. The 3-processor design (design 2 in Section 4.1), which had become inferior, becomes non-inferior again. The availability of high-performance communication links makes multiprocessing more useful again.

4.3.3 Imposition of Arbitrary Constraints

As we mentioned in Section 3.1, SOS can easily handle arbitrary constraints imposed by the designer. Here, we demonstrate how such constraints can be considered, and how they can affect the design.

We performed three experiments. In the first experiment, we imposed a constraint that the input $i_{1,1}$ coming from the external environment becomes available only at time 1. This may be because the data $i_{1,1}$ is supplied by some other system which produces it at time 1, for example. Now, we investigated the best-performance system in this case. The best-performance system synthesized is as follows:

- The system consists of 3 processors: p_{1a} - a processor of type p_1, p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtask S_1, processor p_{2a} performs subtasks S_4, and processor p_{3a} performs subtask S_2 and S_3 in that order. There are two communication links: $l_{1a,2a}$, and $l_{1a,3a}$. Data $i_{4,1}$ gets transmitted on link $l_{1a,2a}$, and data $i_{3,1}$ gets transmitted on link $l_{1a,3a}$.

As one can see, the original best-performance system of 3 processors and 3 communication links has become inferior under this new constraint. Again, intuitively this makes sense. Imposition of the constraint implies an inherent limitation on performance as well as some reduction in parallelism. So, even if we have an expensive system, it would not be exploited to its fullest potential, and in fact, a cheaper system may be able to provide the same performance.

The second experiment involved putting a similar constraint on the input $i_{2,1}$; i.e., $i_{2,1}$ becomes available only at time 1. The best-performance system synthesized in this case is as follows:

- The system consists of 2 processors: p_{1a} - a processor of type p_1, and p_{2a} - a processor of type p_2. Processor p_{1a} performs subtasks S_1 and S_4 in that order, and processor p_{2a} performs subtasks S_2 and S_3 in that order. There is a communication link: $l_{1a,2a}$. Data $i_{3,1}$ gets transmitted on link $l_{1a,2a}$.

As one can see, in this case, 3-processor systems have been completely eliminated, and a 2-processor system has become the best-performance system. Of course, the explanation is same as before.

In the third experiment, the constraint involved the input $i_{4,2}$. However, in this case, the best-performance system remained to be the same as the one without the constraint (i.e., 3 processors, 3 communication links). This is so because the arbitrary constraint in this case did not impose any new constraint effectively. The constraint essentially puts a restriction on the start time of subtask S_4. Since S_4 could not start before time 1 in any case (as it could not start before S_1 is completed), the fact that $i_{4,2}$ does not become available until time 1 does not impose any effective constraint on the start time of S_4. Thus, it does not affect the performance or the system synthesized.

The experiments in this section have clearly demonstrated how arbitrary constraints can affect the system synthesized, and that SOS can deal with this aspect of system design.

		Execution Time			
Proc.	Cost	S_1	S_2	S_3	S_4
p_1	20	24	15	–	18
p_2	19	72	15	144	9
p_3	16	–	45	48	–

Table 4.3: New Processor Characteristics - Example 1

Design	Cost	Performance (time units)
1	69	73
2	43	88
3	42	121
4	19	240

Table 4.4: Example 1 Systems with New Data

4.3.4 New Set of Data

In this section, we use a completely new set of data in the experiments. This is to demonstrate how designs synthesized could be different when the parameter values are varied.

The costs of the processors and the execution times of various subtasks on the processors are given in Table 4.3. The volumes of data that need to be communicated are specified as: $V_{1,3} = 1$, $V_{1,4} = 1$, and $V_{2,3} = 2$. Local transfer delay is given to be negligible; i.e., $D_{CL} = 0$. The cost of a link, C_L, is 7 units; and the remote transfer delay for a unit volume of data over a link, D_{CR}, is one unit.

In this case, 4 non-inferior systems were found. Cost and performance for the four systems are given in Table 4.4.

A discussion of these designs follows:

- *Design 1:* This design consists of 3 processors: p_{1a} - a processor of type p_1, p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Processor p_{1a}

performs subtasks S_1 and S_4 in that order, processor p_{2a} performs subtask S_2, and processor p_{3a} performs subtask S_3. There are two communication links: $l_{1a,3a}$, and $l_{2a,3a}$. Data $i_{3,1}$ gets transmitted on link $l_{1a,3a}$, and data $i_{3,2}$ gets transmitted on link $l_{2a,3a}$. Since S_1 and S_4 are mapped to the same processor, data $i_{4,1}$ does not get transmitted over any link. The whole task is completed at time 73, and the total cost (processors and links) is 69 units.

- *Design 2:* This design is cheaper than the previous one, and consists of 2 processors: p_{1a}, and p_{3a}. There is a link: $l_{1a,3a}$. Processor p_{1a} performs subtasks S_2, S_1 and S_4 in that order, and processor p_{3a} performs subtask S_3. Data $i_{3,2}$ and data $i_{3,1}$ get transmitted on link $l_{1a,3a}$ in that order. Data $i_{4,1}$ does not get transmitted at all. The task completion time is 88, and the total cost is 43 units.

- *Design 3:* This design also consists of 2 processors: p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Processor p_{2a} performs subtasks S_1 and S_4 in that order, and processor p_{3a} performs subtasks S_2 and S_3 in that order. There is a communication link: $l_{2a,3a}$. Data $i_{3,1}$ gets transmitted on link $l_{2a,3a}$. Data $i_{4,1}$ and data $i_{3,2}$ do not get transmitted. The task completion time is 121, and the total cost is 42 units.

- *Design 4:* This design consists of just 1 processor: p_{2a} - a processor of type p_2. The processor performs the subtasks S_1, S_4, S_2, and S_3 in that order. No data transmission takes place, since it's a uniprocessor system. The task completion time is 240, and the total cost is 19 units.

As one can see, the designs synthesized are quite different with the new set of parameter values.

Experiments reported in this section clearly demonstrate SOS's capability in exploring various tradeoffs. SOS can be very effectively used to perform tradeoff studies.

		Execution Time								
Proc.	Cost	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
p_1	4	2	2	1	1	1	1	3	–	1
p_2	5	3	1	1	3	1	2	1	2	1
p_3	2	1	1	2	–	3	1	4	1	3

Table 4.5: Processor Characteristics - Example 2

Design	Runtime (min)	Cost	Performance (time units)
1	62.2	15	5
2	445.17	12	6
3	538.67	8	7
4	75.18	7	8
5	6416.87	5	15

Table 4.6: Example 2 Systems

4.4 Example 2: Nine-Subtask Task Graph

The data flow graph is shown in Figure 4.3. For this example, we assumed that a subtask requires all the inputs before it can start and that none of the outputs from a subtask become available until its execution is over. Again, there are three types of processors, with the costs and the execution times given in Table 4.5. The volume of data is one unit for each arc. We are given: $D_{CL} = 0$, $D_{CR} = 1$, $C_L = 1$.

The MILP model consists of 47 timing and 225 binary variables, and 1081 constraints. We generated 5 non-inferior systems by changing the constraint value for the total system cost, and optimizing the system performance. Bozo's runtime for each of these designs is on the order of a few hours, except for design 5. Cost, performance and runtime for the five designs are given in Table 4.6.

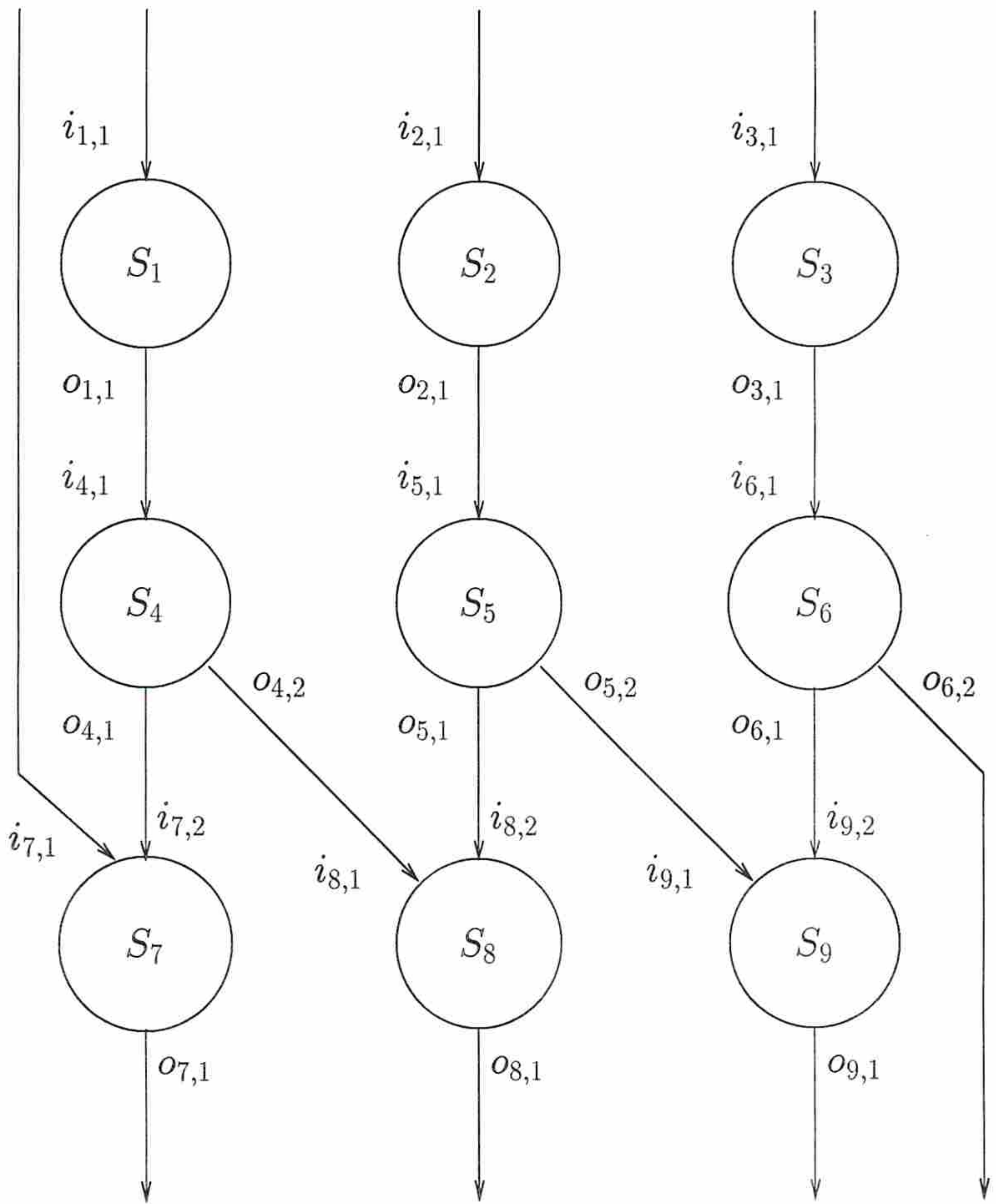


Figure 4.3: Example 2 Task Graph

A brief discussion of the designs follows:

- *Design 1:* This design consists of 3 processors: p_{1a} - a processor of type p_1, p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_3, S_6 and S_4 in that order, processor p_{2a} performs subtasks S_2, S_5, S_9 and S_7 in that order, and processor p_{3a} performs subtasks S_1 and S_8 in that order. There are four communication links: $l_{1a,2a}$, $l_{1a,3a}$, $l_{2a,3a}$, and $l_{3a,1a}$. Data $i_{9,2}$ and data $i_{7,2}$ get transmitted on link $l_{1a,2a}$ in that order, data $i_{8,1}$ gets transmitted on link $l_{1a,3a}$, data $i_{9,1}$ gets transmitted on link $l_{2a,3a}$, and data $i_{4,1}$ gets transmitted on link $l_{3a,1a}$.

- *Design 2:* This design also consists of 3 processors: p_{1a}, p_{1b} - two processors of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_1, S_4 and S_7 in that order, processor p_{1b} performs subtasks S_3, S_6 and S_9 in that order, and processor p_{3a} performs subtasks S_2, S_5 and S_8 in that order. There are two communication links: $l_{1a,3a}$, and $l_{3a,1b}$. Data $i_{8,1}$ gets transmitted on link $l_{1a,3a}$, and data $i_{9,1}$ gets transmitted on link $l_{3a,1b}$.

- *Design 3:* This design consists of 2 processors: p_{1a} - a processor of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_3, S_6, S_4, S_7 and S_9 in that order, and processor p_{3a} performs subtasks S_1, S_2, S_5 and S_8 in that order. There are two communication links: $l_{1a,3a}$, and $l_{3a,1a}$. Data $i_{8,1}$ gets transmitted on link $l_{1a,3a}$, and data $i_{4,1}$ and data $i_{9,1}$ get transmitted on link $l_{3a,1a}$ in that order.

- *Design 4:* This design is similar to design 3, and also consists of 2 processors: p_{1a}, and p_{3a}. However, it has only one link: $l_{1a,3a}$. Presence of only one link forces a change in the mapping between the resources and the events. Processor p_{1a} performs subtasks S_3, S_6, S_1, S_4 and S_7 in that order, and processor p_{3a} performs subtasks S_2, S_5, S_9 and S_8 in that order. Data $i_{9,2}$ and data $i_{8,1}$ get transmitted on link $l_{1a,3a}$ in that order.

- *Design 5:* This design consists of just 1 processor: p_{2a} - a processor of type p_2. The processor performs the subtasks S_2, S_1, S_4, S_5, S_8, S_3, S_7, S_6 and S_9 in that order.

Chapter 5

Bus-Style Interconnection Model

As described in Chapter 3, the SOS approach is quite general and flexible. To support this claim, in this chapter, we demonstrate application of the approach to bus-style interconnection by developing the applicable mathematical programming model. Once again, the mathematical programming model is determined by the task model and the system model used. The task model remains the same as the one described in Section 3.2.1. However, the system model is somewhat different from the one described in Section 3.2.2 to express the fact that bus-style interconnection is being used. These differences are highlighted in the next section. Here an assumption is made that only one bus is allowed in the system; however, it is easy to extend the model to allow several buses as well as different types of buses in the system.

5.1 The System Model

In bus-style interconnection, the system consists of a set of processors and a bus connecting all the processors to each other. So, the cost of the system is dominated by the costs of the processors selected. Thus, the multiprocessor system is specified in terms of the processors selected connected by a common bus. A processor sends data to another processor over the common bus; i.e., all the interprocessor communication takes place by message-passing over the bus.

The necessary data transfers take place over the common bus. If two data transfers are supposed to take place at the same time, then the second can only start after the first is completed. Once again, the waiting time component is captured in the mathematical programming model as a delay in scheduling the communication by enforcing exclusion in the usage of the bus. Once a data transfer operation begins, the bus is released only after the operation is completed; i.e., the bus is busy for an uninterrupted duration of $D_{CR}V_{a1,a2}$ time units if $V_{a1,a2}$ is the volume of data associated with the operation.

Note that only the differences between this system model and the one in Section 3.2.2 are described here. The other aspects of the system model are the same as in Section 3.2.2.

5.2 The Mathematical Programming Model

5.2.1 The Constraints

The variables defined remain the same as in Section 3.2.3.1. Also, the forms of all other constraints remain same except that of the "communication-link-usage-exclusion constraint." The "communication-link-usage-exclusion constraint" gets replaced by the following "bus-usage-exclusion constraint."

Bus-usage-exclusion constraint: If any two data transfers are of remote type, then they both get transmitted over the same bus and exclusion in the usage of the bus must be ensured. For each pair of inputs $i_{a1,b1}$ and $i_{a2,b2}$ (to subtasks S_{a1} and S_{a2} respectively, and from subtasks S_{a3} and S_{a4} respectively), the following relation ensures that the overlap in the usage of the bus by the two data transfers is prevented:

$$\gamma_{a3,a1}\gamma_{a4,a2}L([T_{CS}(i_{a1,b1}), T_{CE}(i_{a1,b1})], [T_{CS}(i_{a2,b2}), T_{CE}(i_{a2,b2})]) = 0$$

5.2.2 Objective Functions

The overall system performance can still be expressed in the same way as described in Section 3.2.3.2. However, the expression for the total system cost will be somewhat different. We do not need to define the "communication-link-creation variables." The system cost can be expressed as:

$$\sum_{d|p_d \in P} \beta_d C_d$$

where C_d is the cost of a processor p_d. The variables of type β_d are related to the variables of type $\sigma_{d,a}$ in the same way as described in Section 3.2.3.2. It would be trivial to extend the model by a constant factor for bus cost, or an incremental bus cost for every processor used in the system.

5.3 Synthesis Using the Model

The newly introduced bus-usage-exclusion constraint is non-linear, and can be easily linearized the way the communication-link-usage-exclusion constraint was linearized in Section 3.2.4 by defining a binary variable $\phi_{a1,b1,a2,b2}$. $\phi_{a1,b1,a2,b2} = 1$ indicates the data transfer for $i_{a1,b1}$ takes place before the data transfer for $i_{a2,b2}$ if both the data transfers are of remote type. Using this variable, the bus-usage-exclusion constraint can be rewritten as the following pair of linear relations:

$$\begin{aligned} T_{CS}(i_{a2,b2}) &\geq T_{CE}(i_{a1,b1}) - (3 - \phi_{a1,b1,a2,b2} - \gamma_{a3,a1} - \gamma_{a4,a2})T_M \\ T_{CS}(i_{a1,b1}) &\geq T_{CE}(i_{a2,b2}) - (2 + \phi_{a1,b1,a2,b2} - \gamma_{a3,a1} - \gamma_{a4,a2})T_M \end{aligned}$$

The linearized MILP model can now be solved using *Bozo*.

Design	Runtime (sec)	Cost	Performance
1	9	14	3.25
2	13	6	4
3	4	5	7

Table 5.1: Systems for Four-Subtask Graph (Bus-Style)

5.4 Experiments and Results

The bus-style interconnection model was also used to experiment with two example task graphs described in Chapter 4.

5.4.1 Example 1: Four-Subtask Task Graph

Processor characteristics and values of all other relevant parameters used in this experiment remained same as given in that Section 4.1.

The MILP bus-architecture model for the four-subtask graph example consists of 21 timing and 43 binary variables, and 117 constraints. Bozo was used to generate 3 non-inferior systems. These different systems were generated by changing the constraint value for the total cost of the system, and optimizing the overall performance of the system. Bozo's runtime to generate each of these designs is on the order of a few seconds. Cost, performance and runtime for the four designs are given in Table 5.1.

A brief discussion of these designs follows:

- *Design 1:* This design consists of 3 processors: p_{1a} - a processor of type p_1, and p_{2a}, p_{2b} - two processors of type p_2. Processor p_{1a} performs subtask S_1, processor p_{2a} performs subtasks S_2 and S_3 in that order, and processor p_{2b} performs subtasks S_4. Data $i_{3,1}$ and data $i_{4,1}$ get transmitted on the common bus in that order.

- *Design 2:* This design consists of 2 processors: p_{1a} - a processor of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_1 and

Design	Runtime (min)	Cost	Performance (time units)
1	107.3	10	6
2	89.53	6	7
3	61.52	5	15

Table 5.2: Systems for Nine-Subtask Graph (Bus-Style)

S_4 in that order, and processor p_{3a} performs subtasks S_2 and S_3 in that order. Data $i_{3,1}$ gets transmitted on the common bus.

- *Design 3:* This design consists of just 1 processor: p_{2a} - a processor of type p_2. The processor performs the subtasks S_2, S_1, S_4, and S_3 in that order.

5.4.2 Example 2: Nine-Subtask Task Graph

Processor characteristics and values of all other relevant parameters used in this experiment remained same as given in that Section 4.4.

The MILP bus-architecture model for the nine-subtask graph example consists of 47 timing and 153 binary variables, and 416 constraints. Three non-inferior systems were generated by changing the constraint value for the total system cost, and optimizing the system performance. Runtime for each of these designs is on the order of a few hours. Table 5.2 gives the statistics for the three designs.

A brief discussion of the designs follows:

- *Design 1:* This design consists of 3 processors: p_{1a}, p_{1b} - two processors of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_1, S_4 and S_7 in that order, processor p_{1b} performs subtasks S_3, S_6 and S_9 in that order, and processor p_{3a} performs subtasks S_2, S_5 and S_8 in that order. Data $i_{8,1}$ and data $i_{9,1}$ get transmitted on the common bus in that order.

- *Design 2:* This design consists of 2 processors: p_{1a} - a processor of type p_1, and p_{3a} - a processor of type p_3. Processor p_{1a} performs subtasks S_3, S_6, S_4, S_7 and S_9 in that order, and processor p_{3a} performs subtasks S_1, S_2, S_5 and S_8 in that order. Data $i_{4,1}$, data $i_{8,1}$ and data $i_{9,1}$ get transmitted on the common bus in that order.

- *Design 3:* This design consists of just 1 processor: p_{2a} - a processor of type p_2. The processor performs the subtasks S_2, S_1, S_4, S_3, S_5, S_8, S_6, S_9 and S_7 in that order.

Chapter 6

Memory Modeling and Other Extensions of the SOS Model

In Chapter 5, we demonstrated our claim about the flexibility of the SOS approach by applying it to bus-style interconnections. In this chapter, we further support this claim by applying it to model memory as an explicit design parameter in system synthesis. Flexibility of the SOS approach is further demonstrated by describing modeling of some complex memory issues. Some of the memory issues considered here and the corresponding SOS models have also been reported in [PP92d].

6.1 Memory Modeling

As the reader might notice, the specific SOS models described in Chapters 3 and 5 do not consider memory costs explicitly. The system models considered assume that each processor has sufficient local memory. However, the cost of the memory is not included in the system cost expression. It is assumed that the costs of memory have already been implicitly included in the costs of the processors. We show how to enhance the SOS model to explicitly include such memory costs next.

6.1.1 Local Memory Costs

The primary function of the processors is to execute the subtasks. Based on the subtasks mapped to a given processor in the synthesized system, the processor must have certain amount of local memory associated with it. The subtasks make use of local memory for two purposes:

- *Code for the subtask:* A subtask requires some local memory at the processor (to which it is mapped) for the code associated with the subtask. The amount of memory required for this purpose could depend on the processor-type involved.

- *Temporary computation data for the subtask:* A subtask could require some local memory space during its execution for temporary data produced. Once again, in general, the amount for this purpose could also depend on the processor-type involved. Note that any static data associated with the subtask is considered to be part of the code memory.

Here, we show how to enhance the system cost expression to include the cost associated with the above memory requirements.

6.1.1.1 The Task Model and the System Model

A subtask S_a requires a certain amount of code memory depending on the processor type on which it is performed. A parameter, denoted as $M_{CPS}(P_t, S_a)$, specifies the code memory requirement for the subtask S_a if processor type P_t is selected to perform it. Similarly, the subtask requires certain amount of temporary data memory; and a parameter, denoted as $M_{DPS}(P_t, S_a)$, specifies the said requirement for the subtask S_a if processor type P_t is selected to perform it. Each processor in the system has a certain amount of local memory. A subtask S_a executing on a processor p_d uses its local memory for code as well as temporary data. The local memory associated with p_d is occupied by the temporary data associated with S_a during the period p_d is occupied with the execution of

S_a and that period only. However, the code associated with S_a is permanently resident in the local memory of p_d. The cost of a unit amount of memory is C_M.

All other aspects of the task model and the system model remain the same as described in Chapter 3 (for point-to-point interconnection) or Chapter 5 (for bus-style interconnection).

6.1.1.2 The Mathematical Programming Model

In the synthesized system, each processor will have a certain amount of local memory depending upon the subtasks that are to be executed on the processor. So, the cost of the system obviously depends on the amounts of memory present at the different processors in the system. Thus, one of the components of the total cost of the system will now be the cost of the total amount of memory present in the system. In order to write an expression for the total amount of memory, we need to define a *memory variable* M_d for each processor $p_d \in P$, where M_d represents the amount of memory present at the processor p_d. Using this variable, the cost of the total amount of memory present in the system can be expressed as:

$$\sum_{d|p_d \in P} M_d C_M$$

The above expression will get added to the other components of the system cost to derive the total system cost. The variables of type M_d are related to the variables of type $\sigma_{d,a}$, and the parameters $M_{CPS}(P_t, S_a)$ and $M_{DPS}(P_t, S_a)$. In order to express these relationships concisely, let us represent the amount of local memory at a processor p_d as the sum of two components: code memory (denoted as $M_{C,d}$) and data memory (denoted as $M_{D,d}$); i.e.,

$$M_d = M_{C,d} + M_{D,d} \tag{6.1}$$

The code memory component will be the sum of the code memory requirements of all the subtasks mapped to the processor (as the codes are permanently resident in the local memory); i.e.,

$$M_{C,d} = \sum_{a|p_d \in P_a} \sigma_{d,a} M_{CPS}(Typ(p_d), S_a) \tag{6.2}$$

where $Typ(p_d)$ represents the type of the processor p_d. The data memory component will be equal to the largest amount (of temporary data memory) that is required by any of the subtasks mapped to the processor (as the temporary data space gets freed up as soon as a subtask is completed), which can be expressed by introducing the following constraint in the model (for all a such that $p_d \in P_a$):

$$M_{D,d} \geq \sigma_{d,a} M_{DPS}(Typ(p_d), S_a) \tag{6.3}$$

All other aspects of the mathematical programming model remain the same as described in Chapter 3 (for point-to-point interconnection) or Chapter 5 (for bus-style interconnection).

6.1.1.3 Synthesis Using the Model

The linearized MILP model corresponding to the mathematical programming model developed (with memory costs included), when solved using *Bozo*, produces the following information as outputs:

- a multiprocessor system; i.e., the chosen set of processors and the interconnection architecture,
- a schedule for the subtasks,
- detailed timing information for computation and transfer of data, and
- the amount of local memory present at each of the selected processors.

Subtask	S_1	S_2	S_3	S_4
Memory Amount	1	2	3	4

Table 6.1: Memory Requirements for Four-Subtask Graph

6.1.2 Experiments and Results

Here, we describe the results of some experiments, including local memory costs, with the extended model described in the previous section. In order to be able to understand the role of inclusion of memory as an explicit design parameter in the system synthesis process, we compare the results here with the results obtained when memory costs were not explicitly considered. In this comparison, we refer to the model considering memory costs as "extended model" while the one not considering such costs as "original model." Both the example 1 and example 2 task graphs described in Chapter 4 were used in the experiments.

Here, we assume that the memory requirements of the subtasks are the same for all the processor types involved. Under this assumption, the contribution to the system cost due to the memory requirements for the subtask codes becomes a constant; i.e, it is independent of the mapping between subtasks and processors. Hence, code memory requirements can be ignored in the system design and the tradeoff studies; so we only consider data memory requirements in the following.

6.1.2.1 Example 1: Four-Subtask Task Graph

This is the same example as the one in Section 4.1. Here, we include the memory requirements of the subtasks. Table 6.1 lists the amounts of temporary data memory required during the execution of each of the subtasks. The cost of a unit amount of memory, C_M, is given to be one unit. All the other parameters remain as in Section 4.1. For this graph, we performed synthesis (including memory costs) for bus-style interconnection as well as point-to-point interconnection.

Design	Runtime (sec)	Cost	Performance
1	43	22	3.25
2	24	13	4
3	38	9	7

Table 6.2: Bus-Style Systems for Four-Subtask Graph (With Memory)

Bus-Style Interconnection: Here, the "original model" refers to the model described in Chapter 5; and the designs found using the original model refer to the designs presented in Section 5.4.1.

The extended MILP bus-architecture model for the example consists of 21 timing, 43 binary and 9 memory variables, and 136 constraints. 3 non-inferior systems were found using Bozo (by changing the constraint value for the total cost of the system, and optimizing the overall performance of the system). Bozo's runtime for each of the designs is again on the order of a few seconds. Cost, performance and runtime for the three designs are given in Table 6.2.

A brief discussion of the designs follows:

- *Design 1:* This is same as design 1 found using the original model. The amounts of local memory at processors p_{1a}, p_{2a}, and p_{2b} are 1, 3, and 4 units respectively.

- *Design 2:* This is same as design 2 found using the original model. The amounts of local memory at processors p_{1a} and p_{3a} are 4 and 3 units respectively.

- *Design 3:* This is same as design 3 found using the original model. The amount of local memory at processor p_{2a} is 4 units.

These experiments indicate that memory can indeed be modeled as an explicit design parameter in the SOS approach.

Design	Runtime (sec)	Cost	Performance (time units)
1	12	22	2.5
2	32	14	4
3	66	9	7

Table 6.3: Point-to-Point Systems for Four-Subtask Graph (With Memory)

Point-to-Point Interconnection: Here, the "original model" refers to the model described in Chapter 3; and the designs found using the original model refer to the designs presented in Section 4.1.

The extended MILP point-to-point model for the example consists of 21 timing, 72 binary and 9 memory variables, and 193 constraints. However, only 3 non-inferior systems were found using Bozo in this case (again by changing the constraint value for the total cost of the system, and optimizing the overall performance of the system). Bozo's runtime for each of the designs is again on the order of a few seconds. Cost, performance and runtime for the three designs are given in Table 6.3.

A brief discussion of the designs follows:

- *Design 1:* This design is essentially the same as design 1 found using the original model. The amounts of local memory at processors p_{1a}, p_{2a}, and p_{3a} are 1, 4, and 3 units respectively.

- *Design 2:* This design is essentially the same as design 3 found using the original model. The amounts of local memory at processors p_{1a} and p_{3a} are 4 and 3 units respectively.

- *Design 3:* This design is essentially the same as design 4 found using the original model. The amount of local memory at processor p_{2a} is 4 units.

Note that a 3-processor design (design 2 using the original model was non-inferior when memory costs were not considered) becomes inferior when memory costs are explicitly included in the extended model.

Subtask	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
Memory Amount	1	2	3	4	5	6	7	8	9

Table 6.4: Memory Requirements for Nine-Subtask Graph

6.1.2.2 Example 2: Nine-Subtask Task Graph

This is the same example as the one in Section 4.4. Table 6.4 lists the amounts of temporary data memory required during the execution of each of the subtasks. The cost of a unit amount of memory $C_M = 1$. All the other parameters remain as in Section 4.4.

For this example, we performed system synthesis with the bus-style interconnection model (of course, including memory costs as well). Hence, the "original model" in this section refers to the model described in Chapter 5; and the designs found using the original model refer to the designs presented in Section 5.4.

The extended MILP bus-architecture model for the example consists of 47 timing, 153 binary and 9 memory variables, and 468 constraints. However, seven non-inferior systems were found in this case. These different systems were generated by changing the constraint value for the system performance, and optimizing the total system cost. Runtime for these designs is much higher on the average (compared to Table 5.2). The higher runtime seems attributable to the fact that the design space exploration in this case was done by optimizing the total system cost while specifying the system performance as a constraint. In general, it has been observed during various experiments performed with SOS that the runtimes are lower when the exploration is done by optimizing the system performance while specifying the total system cost as a constraint. Table 6.5 gives the statistics for the seven designs.

A brief discussion of the designs follows:

- *Design 1:* This design consists of 3 processors: p_{1a} - a processor of type p_1, p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Subtasks

Design	Runtime (min)	Cost	Performance (time units)
1	109.87	28	6
2	109.44	23	7
3	2054.21	22	8
4	5281.01	21	10
5	3945.4	18	11
6	2300.07	17	12
7	59.69	14	15

Table 6.5: Bus-Style Systems for Nine-Subtask Graph (With Memory)

S_3, S_6, S_4 and S_7 are mapped to processor p_{1a}, subtasks S_2, S_5, S_9 and S_8 are mapped to processor p_{2a}, and subtask S_1 is mapped to processor p_{3a}. The amounts of local memory at processors p_{1a}, p_{2a}, and p_{3a} are 7, 9, and 1 unit(s) respectively.

- *Design 2:* This design is essentially the same as design 2 found using the original model. The amounts of local memory at processors p_{1a}, and p_{3a} are 9, and 8 units respectively.
- *Design 3:* This design is similar to design 2. However, the cost of the system has been improved by reducing the amount of memory required. This has been accomplished by changing the mapping between subtasks and processors. Subtasks S_1, S_3, S_4, S_6 and S_7 are mapped to processor p_{1a}, and subtasks S_2, S_5, S_8 and S_9 are mapped to processor p_{3a}. This mapping requires 7 units of memory at processor p_{1a}, and 9 units at processor p_{3a}.
- *Design 4:* This design also consists of 2 processors: p_{2a} - a processor of type p_2, and p_{3a} - a processor of type p_3. Subtasks S_3, S_4, S_6, S_7, S_8 and S_9 are mapped to processor p_{2a}, and subtasks S_1, S_2 and S_5 are mapped to processor p_{3a}. The amounts of local memory at processors p_{2a} and p_{3a} are 9 and 5 units respectively.

- *Design 5:* This design is similar to design 4. However, the amount of memory required has been reduced by moving the subtask S_5 from processor p_{3a} to processor p_{2a}. This new mapping requires 9 units of memory at processor p_{2a}, and only 2 units at processor p_{3a}.

- *Design 6:* This design is similar to design 5. However, the amount of memory required has been further reduced by moving the subtask S_2 from processor p_{3a} to processor p_{2a}. The new mapping requires 9 units of memory at processor p_{2a}, and only 1 unit at processor p_{3a}.

- *Design 7:* This design is essentially the same as design 3 found using the original model. The amount of local memory at processor p_{2a} is 9 units.

An interesting point to note about these designs is how several new non-inferior designs have been added by varying the mapping between subtasks and processors (and thus varying the memory requirements) while keeping the same set of processors in the system. Also note that the number of 2-processor designs found with the extended model is much more than that with the original model, whereas the number of 3-processor designs found is the same with the two models.

The observations made from the experimental results with the four-subtask graph and the nine-subtask graph seem to suggest that the systems with larger number of processors would tend to become inferior when memory costs are considered, and that systems with fewer numbers of processors would tend to provide better overall cost-performance ratio. This observation seems in tune with the intuitive expectation. Each processor in the system requires a certain amount of local memory; so the larger the number of processors, the larger the total amount of memory required in the system. Hence, it is conceivable that systems with larger number of processors would become inferior (because of high overall cost), and that systems with fewer number of processors would provide better cost-performance ratio.

Of course, the experiments clearly indicate that in order to obtain high-quality designs, it is important to include memory costs explicitly in the model while designing the system.

6.1.3 Memory Buffers for Communication

In Section 6.1.1, we showed how to model some of the memory costs explicitly. Primarily, we looked at memory requirements during the computation of the subtasks. Here, we investigate how one could consider the memory requirements due to the interprocessor communication aspects of the overall task execution.

As one would recall, in the SOS models of Chapter 3 and 5, it is assumed that data transfer operations are taken care of by I/O modules. While executing the subtasks, processors produce the data to be transferred to other processors in the system. Once the data to be transferred has been produced, it will actually be transferred to the processor requiring it by an I/O module over the appropriate communication link. It is also assumed that necessary I/O modules are present in the system. It is easy to see that in the point-to-point interconnection model, an I/O module is associated with each point-to-point communication link; whereas in the bus-style interconnection model, a single I/O module (or a bus controller) is sufficient for the whole system (as at a time, only one data transfer operation can be performed over the common bus). Now, if I/O modules are performing the data transfer operations, it is not difficult to visualize that one would require certain amount of memory buffers for communication associated with each of the processors. First, the processor producing the data to be transferred outputs it, and places it in some *output buffer* for the I/O module to read it from. Then the appropriate I/O module reads it from the output buffer and transfers it to the receiving processor. As the receiving processor may not use the transferred data immediately, the I/O module places it in another buffer associated with the receiving processor called *input buffer*. Thus, each processor has two kinds of communication buffers associated with it:

- *Input buffer:* This is where the data received by the processor is placed by the I/O module, and is later accessed by the processor at the appropriate time.

- *Output buffer:* This is where the data to be transferred to another processor is placed by the processor, and is later transferred by the appropriate I/O module to the processor using it.

Now, in order to explicitly include the costs associated with such communication buffers in the system cost expression, we need to figure out the amounts of such buffers at each of the processors in the system.

6.1.3.1 Input Buffer

Let us attempt to figure out the number of input buffers or amount of input buffer space required at a processor p_d in the synthesized system. Let the number be represented by $M_{I,d}$.

As we said before, the input buffer space is used for storing the data sent from other processors and used by the processor p_d as input. So, the input buffer amount required at the processor p_d depends on the input data required from other processors, which in turn depends on the subtasks mapped to p_d. If a subtask S_a is mapped to the processor p_d, and one of subtask's inputs $i_{a,b}$ is coming from a subtask mapped to another processor, then the data corresponding to $i_{a,b}$ will obviously get stored in the input buffer of p_d. In other words, if the input $i_{a,b}$ required at the processor p_d is remotely transferred, we require space for it in the input buffer. Say, the input $i_{a,b}$ is actually supplied by the output $o_{a1,c}$ from subtask S_{a1}. In that case, the fact that the input $i_{a,b}$ is remotely transferred, is expressed by the binary variable $\gamma_{a1,a}$; and the fact that the input $i_{a,b}$ is required at the processor p_d is expressed by the binary variable $\sigma_{d,a}$. The volume of data associated with the input $i_{a,b}$ is $V_{a1,a}$. Thus, if the input $i_{a,b}$ is required at the processor p_d and it is remotely transferred, then at least we

require $V_{a1,a}$ amount of input buffer space. Mathematically, this can be expressed as the following constraint in the model:

$$M_{I,d} \geq V_{a1,a}\sigma_{d,a}\gamma_{a1,a} \tag{6.4}$$

We need a constraint of the above form for all $a1, a$ such that $p_d \in P_a$ and subtask S_{a1} sends data to subtask S_a; that would ensure all the inputs like $i_{a,b}$ (i.e., required at p_d and remotely transferred) have been accounted for. The set of constraints expressed by Equation 6.4 essentially says that the amount of input buffer space required at p_d is at least equal to the largest volume of input data required by a subtask mapped to it and remotely transferred to it. This is the case since in the static scheduling approach used, it is ensured that any input data remotely transferred to a subtask is transferred and put in the input buffer before the subtask actually requires the data during its computation.

Obviously, Equation 6.4 is non-linear. However, it is easy to linearize. Let us define a large constant V_M which represents the sum of volumes of data at all the arcs present in the given task graph. Equation 6.4 can then be rewritten as the following linear relation:

$$M_{I,d} \geq V_{a1,a} - (2 - \sigma_{d,a} - \gamma_{a1,a})V_M \tag{6.5}$$

Having written Equation 6.4, now the question is if that constraint accurately reflects the input buffer requirement. Well, the answer is unfortunately no. Definitely, the minimum amount of buffer space required is expressed by the constraint, as the buffer must be able to accommodate any input data volume that is to be stored in the buffer. However, what happens if two of such input data need to be accommodated in the buffer at the same time? Obviously, if such is the case, the amount of buffer space required will be at least the sum of the volumes of the corresponding inputs. Essentially, we need to determine what inputs (to be stored in the buffer) overlap in their *lifetimes* in the buffer, where lifetime of an input data in the buffer is the time-interval during which the input

occupies the buffer. Now, the question is what exactly is the lifetime of an input $i_{a,b}$ in the buffer. We know the time at which $i_{a,b}$ starts getting transferred to the buffer is $T_{CS}(i_{a,b})$. When exactly does the input $i_{a,b}$ stop being present in the buffer? It is not explicit in the model developed so far. The time at which $i_{a,b}$ can be removed from the buffer is the time at which the subtask S_a is finished using it. This aspect has not been modeled so far; we have talked about the time at which a subtask starts requiring an input (f_R parameters), but not the time at which it stops requiring it. So, in order to model this aspect, we need to define another parameter $f_F(i_{a,b})$ associated with each input $i_{a,b}$, which specifies that the input $i_{a,b}$ is no longer required when $f_F(i_{a,b})$ fraction of the subtask S_a is completed. Now, we define another timing variable $T_{IF}(i_{a,b})$ which represents the time when the subtask S_a has finished using the input $i_{a,b}$ and thus no longer requires it. The following relation expresses the constraint on the value of $T_{IF}(i_{a,b})$:

$$T_{IF}(i_{a,b}) = T_{SS}(S_a) + f_F(i_{a,b})(T_{SE}(S_a) - T_{SS}(S_a)) \tag{6.6}$$

Thus, we know the time at which $i_{a,b}$ can be removed from the input buffer. Now, the lifetime of input $i_{a,b}$ in the buffer can be expressed as the time-interval $[T_{CS}(i_{a,b}), T_{IF}(i_{a,b})]$.

Now, let us consider two inputs $i_{a1,b1}$ (supplied by subtask S_{a3} to subtask S_{a1}) and $i_{a2,b2}$ (supplied by subtask S_{a4} to subtask S_{a2}) which are to be stored in the input buffer of processor p_d and potentially overlap in their lifetimes in the buffer. Overlap in the lifetimes can be expressed using the overlap function defined in Section 3.2.3.1 as follows:

$$L([T_{CS}(i_{a1,b1}), T_{IF}(i_{a1,b1})], [T_{CS}(i_{a2,b2}), T_{IF}(i_{a2,b2})])$$

Of course, the fact that $i_{a1,b1}$ and $i_{a2,b2}$ are to be stored in the input buffer of p_d can be expressed by the products $(\sigma_{d,a1}\gamma_{a3,a1})$ and $(\sigma_{d,a2}\gamma_{a4,a2})$ respectively.

If the two lifetimes do indeed overlap, then the two inputs are concurrently present in the buffer and hence the amount of input buffer space should be at least $(V_{a3,a1} + V_{a4,a2})$. In summary, the fact that inputs $i_{a1,b1}$ and $i_{a2,b2}$ potentially overlap in their lifetimes leads to the following constraint on the amount of input buffer space:

$$\begin{aligned} M_{I,d} \geq\ & (V_{a3,a1} + V_{a4,a2})\sigma_{d,a1}\gamma_{a3,a1}\sigma_{d,a2}\gamma_{a4,a2} \\ & L([T_{CS}(i_{a1,b1}), T_{IF}(i_{a1,b1})], [T_{CS}(i_{a2,b2}), T_{IF}(i_{a2,b2})]) \end{aligned} \tag{6.7}$$

Let us consider linearization of Equation 6.7. First, we need to express the "overlap function component" of the equation in a linear form. There is no overlap in the two lifetimes if and only if either *(i)* transfer of data $i_{a1,b1}$ starts after subtask S_{a2} is finished using data $i_{a2,b2}$ or *(ii)* transfer of data $i_{a2,b2}$ starts after subtask S_{a1} is finished using data $i_{a1,b1}$. Let us represent case (i) by the complement of the value of a new binary variable $\psi_{a1,b1,a2,b2}$; i.e., $\psi_{a1,b1,a2,b2} = 0$ implies the transfer of data $i_{a1,b1}$ starts after data $i_{a2,b2}$ is no longer required in the buffer. Similarly, another binary variable $\psi_{a2,b2,a1,b1}$ represents case (ii); i.e., $\psi_{a2,b2,a1,b1} = 0$ implies the transfer of data $i_{a2,b2}$ starts after data $i_{a1,b1}$ is no longer required in the buffer. The following pair of linear equations can be used to express the relationships between the new binary variables and the appropriate timing variables in the overlap function:

$$T_{IF}(i_{a2,b2}) - T_{CS}(i_{a1,b1}) \leq \psi_{a1,b1,a2,b2}T_M \tag{6.8}$$

$$T_{IF}(i_{a1,b1}) - T_{CS}(i_{a2,b2}) \leq \psi_{a2,b2,a1,b1}T_M \tag{6.9}$$

where T_M is the same as the large constant defined in Section 3.2.4.1 (a constant whose value is larger than the possible values of all the timing variables in the model). Given the above pair of relations, there is no overlap in the two lifetimes

if and only if either (i) $\psi_{a1,b1,a2,b2} = 0$ or (ii) $\psi_{a2,b2,a1,b1} = 0$. Thus the overlap function $L([T_{CS}(i_{a1,b1}), T_{IF}(i_{a1,b1})], [T_{CS}(i_{a2,b2}), T_{IF}(i_{a2,b2})])$ can be expressed as the product $(\psi_{a1,b1,a2,b2}\psi_{a2,b2,a1,b1})$. Hence, Equation 6.7 can be rewritten as follows:

$$M_{I,d} \geq (V_{a3,a1} + V_{a4,a2})\sigma_{d,a1}\gamma_{a3,a1}\sigma_{d,a2}\gamma_{a4,a2}\psi_{a1,b1,a2,b2}\psi_{a2,b2,a1,b1} \quad (6.10)$$

However, the above relation is still in a non-linear form. The following is a linear form of the same relation:

$$M_{I,d} \geq (V_{a3,a1}+V_{a4,a2})-(6-\sigma_{d,a1}-\gamma_{a3,a1}-\sigma_{d,a2}-\gamma_{a4,a2}-\psi_{a1,b1,a2,b2}-\psi_{a2,b2,a1,b1})V_M \quad (6.11)$$

where V_M is the same large constant as defined earlier in this section. Thus, the non-linear constraint represented by Equation 6.7 has been expressed in a linear form using the set of linear Equations 6.11, 6.8, 6.9.

6.1.3.2 Output Buffer

In this section, we attempt to determine the amount of output buffer required at a processor p_d in the synthesized system (similar to the attempt for input buffer in previous section). Let us represent the amount of output buffer required by $M_{O,d}$.

The output buffer space is used for storing the data produced by the processor p_d and to be sent to other processors. Hence, the amount of output buffer required depends on the output data to sent to other processors, which once again depends on the subtasks mapped to p_d. If a subtask S_a is mapped to the processor p_d, and one of subtask's outputs $o_{a,c}$ is to go to a subtask mapped to another processor, then the data corresponding to $o_{a,c}$ will obviously get stored in the output buffer of p_d. In other words, if the output $o_{a,c}$ produced at the processor p_d is remotely

transferred, we require space for it in the output buffer. Say, the input $i_{a1,b}$ to subtask S_{a1} is actually supplied by the output $o_{a,c}$. Then the binary variable $\gamma_{a,a1}$ expresses the fact that the output $o_{a,c}$ is remotely transferred to the input $i_{a1,b}$. The volume of data associated with the output $o_{a,c}$ is $V_{a,a1}$. Thus, if the output $o_{a,c}$ is produced at the processor p_d and is remotely transferred, then at least we require $V_{a,a1}$ amount of output buffer space; which mathematically introduces the following constraint in the model:

$$M_{O,d} \geq V_{a,a1}\sigma_{d,a}\gamma_{a,a1} \tag{6.12}$$

A constraint of the above form is required for all $a, a1$ such that $p_d \in P_a$ and subtask S_a sends data to subtask S_{a1}. This set of constraints essentially says that the amount of output buffer space required at p_d is at least equal to the largest volume of output data produced by a subtask mapped to it and remotely transferred to a subtask mapped to some other processor. Using the large constant V_M, the linear form of Equation 6.12 is the following:

$$M_{O,d} \geq V_{a,a1} - (2 - \sigma_{d,a} - \gamma_{a,a1})V_M \tag{6.13}$$

Once again, the question is if the constraint just described accurately reflects the output buffer requirement; and the answer too once again is no. Of course, the buffer must be able to accommodate any output data volume to be stored in the buffer; but what happens if two of such output data need to be accommodated in the buffer at the same time? Obviously, in that case, the amount of buffer space required will be at least the sum of the volumes of the corresponding outputs. Once again, we need to figure out what outputs (to be stored in the buffer) overlap in their lifetimes in the buffer; which brings up the question of what exactly is the lifetime of an output $o_{a,c}$ in the buffer. The time at which $o_{a,c}$ is produced is $T_{OA}(o_{a,c})$. If the output $o_{a,c}$ supplies the input $i_{a1,b}$, then the time at which data $o_{a,c}$ gets transferred to the subtask requiring it, is $T_{CE}(i_{a1,b})$. Hence, the output

$o_{a,c}$ occupies the output buffer during the time-interval $[T_{OA}(o_{a,c}), T_{CE}(i_{a1,b})]$, and that indeed represents its lifetime in the buffer.

Now, let us consider two outputs $o_{a1,c1}$ (produced by subtask S_{a1} and say, supplying input $i_{a3,b3}$ to subtask S_{a3}) and $o_{a2,c2}$ (produced by subtask S_{a2} and say, supplying input $i_{a4,b4}$ to subtask S_{a4}) to be stored in the output buffer of processor p_d that potentially overlap in their lifetimes in the buffer. The fact that outputs $o_{a1,c1}$ and $o_{a2,c2}$ potentially overlap in their lifetimes leads to the following constraint on the amount of output buffer:

$$\begin{aligned} M_{O,d} \geq\ & (V_{a1,a3} + V_{a2,a4})\sigma_{d,a1}\gamma_{a1,a3}\sigma_{d,a2}\gamma_{a2,a4} \\ & L([T_{OA}(o_{a1,c1}), T_{CE}(i_{a3,b3})], [T_{OA}(o_{a2,c2}), T_{CE}(i_{a4,b4})]) \qquad (6.14) \end{aligned}$$

Linearization of Equation 6.14 can be accomplished in a manner similar to that for Equation 6.7. Let us define two new binary variables $\theta_{a1,c1,a2,c2}$ and $\theta_{a2,c2,a1,c1}$. $\theta_{a1,c1,a2,c2} = 0$ implies data $o_{a1,c1}$ is produced after data $o_{a2,c2}$ has been transferred to the input $i_{a4,b4}$. Similarly, $\theta_{a2,c2,a1,c1} = 0$ implies data $o_{a2,c2}$ is produced after data $o_{a1,c1}$ has been transferred to the input $i_{a3,b3}$. Only in these two cases, there is no overlap in the two lifetimes under consideration. Thus, the overlap in the lifetimes can be expressed by the product $(\theta_{a1,c1,a2,c2}\theta_{a2,c2,a1,c1})$, where $\theta_{a1,c1,a2,c2}$ and $\theta_{a2,c2,a1,c1}$ satisfy the following constraints:

$$T_{CE}(i_{a4,b4}) - T_{OA}(o_{a1,c1}) \leq \theta_{a1,c1,a2,c2} T_M \qquad (6.15)$$

$$T_{CE}(i_{a3,b3}) - T_{OA}(o_{a2,c2}) \leq \theta_{a2,c2,a1,c1} T_M \qquad (6.16)$$

Now, Equation 6.14 can be rewritten as follows:

$$M_{O,d} \geq (V_{a1,a3} + V_{a2,a4})\sigma_{d,a1}\gamma_{a1,a3}\sigma_{d,a2}\gamma_{a2,a4}\theta_{a1,c1,a2,c2}\theta_{a2,c2,a1,c1} \tag{6.17}$$

whose linear form (using the large constant V_M) is:

$$M_{O,d} \geq (V_{a1,a3}+V_{a2,a4})-(6-\sigma_{d,a1}-\gamma_{a1,a3}-\sigma_{d,a2}-\gamma_{a2,a4}-\theta_{a1,c1,a2,c2}-\theta_{a2,c2,a1,c1})V_M \tag{6.18}$$

Thus, the set of Equations 6.18, 6.15, 6.16 express the non-linear Equation 6.14 in a linear form.

Chapter 7

Making SOS More Practical

The strength and generality of the SOS approach has been clearly demonstrated in Chapters 3, 4, 5, and 6 by showing how to apply it to different design situations (along with several experimental results). In this chapter, another aspect of the approach is addressed: practicality.

7.1 Some Comments on the Practicality of SOS

By looking at the experimental results discussed in Chapters 4, 5, and 6, it is clear that as we would increase the number of subtasks and/or the number of processor-types, the size of the MILP model could grow fairly quickly, and solving such a large MILP model could become impractical. However, we find applications which require heterogeneous systems and the number of subtasks and processor-types involved are relatively small. As an example, in [MT82], the problem of optimum power flow in power systems has been modeled as a task graph consisting of 8 subtasks, and the system to solve the problem has been designed using 4 types of processors. The SOS approach can definitely be used to perform synthesis for such applications.

As we go to larger applications, the amount of computer time to solve the MILP model could grow. However, we believe that the approach may still be

usable since it ensures an optimal design in return. A human designer may actually require longer design time than the computer time required by SOS, if it is necessary to ensure the optimality of the design. The human designer may still not be sure about the optimality and the correctness of the design. However, as we continue to increase the application size, it is obvious that a point will be reached when the computer time to solve the MILP model is prohibitive. In other words, for very large applications, the MILP approach may not be directly feasible. Nevertheless, the development of the MILP model for the problem is still of great significance, as it provides a complete in-depth model for the synthesis problem. There are two ways to continue with the MILP approach at that point:

- Devise better solution strategies for the MILP model to reduce the runtime.
- Develop some theory using the understanding gained from the MILP model and use it in building algorithmic/heuristic procedures for the synthesis problem.

In this chapter, we concentrate on runtime improvement for solving the MILP model. In Chapter 9, we look at some heuristic ideas for solving the problem.

7.2 Runtime Improvement for the MILP Model

We are using Bozo to solve the MILP model. A detailed description of how Bozo works can be found in [HH90]. For the purposes of our discussion of how to improve the runtime for solving our MILP models using Bozo, we provide the necessary description here.

Essentially, Bozo implements a *Linear Programming-based branch-and-bound algorithm*[7.1] for solving the MILP problems. The details and the fundamental

[7.1] Subsequently referred to as "LP-based branch-and-bound algorithm"

principles underlying such an algorithm can be found in [GN72]. In a very rudimentary form, the following steps are involved:

STEP 1: (Initialization) Read the MILP problem; Solve the initial Linear Programming Relaxation (LPR); If integer solution found, print the solution and stop; Set $Incumbent = -\infty$ for a MAX problem.

STEP 2: (Branching) Pick a fractional-valued binary variable and force it to 0 and 1, creating two subproblems.

STEP 3: (LPR Solution) If no subproblem left, print the *Incumbent* and stop; Else pick a subproblem and solve its LPR.

STEP 4: (Fathoming[7.2] by Integrality) If integer solution found, fathom the subproblem; If the new integer solution better than *Incumbent*, make new solution *Incumbent*; If subproblem fathomed, Go to Step 3.

STEP 5: (Fathoming by Bounds) If LPR solution worse than *Incumbent*, fathom the subproblem and Go to Step 3; Else Go to Step 2.

The phrase "fractional-valued binary variable" means a variable that assumes a non-integer value between 0 and 1 in the LPR solution to the current subproblem, but must assume a binary value in a feasible solution to the given MILP problem. Bozo invokes a commercial linear programming package, XLP, developed by XMP Software, Inc. to solve the LPR. The package uses the *Simplex algorithm* to solve the LPR. The algorithm is fairly commonly used to solve linear programming problems, and its description can be found in [Chv83]. For the purposes of discussion here, it is sufficient to know that the algorithm starts with an initial feasible solution for the LP problem, and then uses an iterative-improvement approach to come up with an optimal solution. At each iteration, it attempts to improve the objective function value. Each such iteration is known as a *simplex pivot*.

[7.2]Fathoming a subproblem implies the subproblem is not going to be explored any further. This happens when the subproblem's LPR solution is either an integer solution or worse than *Incumbent*.

Within the general framework of the branch-and-bound algorithm which is essentially an exploration of the search space containing all feasible solutions to the MILP problem, two intuitive ways to improve the runtime are:

- Reduce the search space itself.
- Make the exploration more efficient, and thus avoid unnecessary search.

Reduction of the search space involves generation of better bounds on various design parameters and variables of interest. An attempt is made in this direction in Chapter 8 where some theory is developed and associated difficulties are addressed.

In this chapter, we concentrate on the idea of making the exploration more efficient in order to improve the runtime for solution of MILP model. Given that the subject of "bounds" is addressed in Chapter 8, with respect to the idea of making exploration more efficient, we need to address the subjects of "branching strategy" (step 2) and "LPR solution technique" (step 3) here. Branching strategies are discussed in the next section.

As far as the LPR solution technique is concerned, as we said before, the simplex algorithm is being used. The algorithm is an efficient algorithm and is known to perform well in practice. There is not a strong possibility of a gain by improvising the algorithm here. However, another weakness of the overall LP-based branch-and-bound algorithm stems from this aspect that a LPR solution technique is employed to gauge the feasibility and/or potential of a subproblem and hence those of the branching decision that lead to the subproblem. As we know, solution of an MILP problem involves optimization of an objective function while satisfying a set of constraints. In our branch-and-bound algorithm for solving the MILP problem, the check whether the subproblem under consideration (and hence the set of decisions leading to that subproblem) is still satisfying the given set of constraints for the MILP problem, is implicitly performed in the LPR solution step. However, this way of constraint satisfaction verification

is not very rigorous. In Section 7.4, we elaborate more on this "flexible" constraint satisfaction check and discuss how this weakness of the algorithm could be strengthened.

7.3 Branching Strategies

As we saw, branching implies selecting a binary variable whose value is fractional (non-binary) in the LPR solution to a given subproblem and forcing it to be binary (0 or 1) in the descendant subproblems. Now, there will usually be more than one fractional-valued binary variable in the LPR solution and the question is which one to select for branching. The answer is provided by the branching strategy used.

Forcing a binary variable to a specific binary value means making a decision about the design. For example, in our model, forcing a binary variable $\sigma_{d,a}$ to be 1 is equivalent to taking a decision that the subtask S_a will be executed on the processor p_d. So, branching on a binary variable is same as making a decision represented by that variable. A branching strategy translates into some sort of ordering of design decisions. It would be desirable to perform branching in the "right" order so that backtracking is minimized. Usually, when an expert designer synthesizes a design, (s)he makes various design decisions in a certain order. It may be effective to use the same order for branching. This can be accomplished by assigning relative priorities to the binary variables, reflecting the order of decisions. However, sometimes a single design decision may be encoded by several binary variables in the model. In such a case, the branching strategy should allow branching on a group of variables.

7.3.1 Assignment of Priorities to Binary Variables

7.3.1.1 Motivation

Binary variables can be assigned relative priorities to reflect the preferred decision order, and these priorities can be taken into account while branching. The relative priorities could be dictated by several factors. For example, if one decision (variable x) depends on another (variable y), in the sense that one can not be made intelligently before the other has been decided, then the variable y will have higher priority than the variable x at the time of branching. Another possible reason for having a high priority for a particular variable x is that fixing x strengthens the constraint system; in such a case it may be important to fix the value of x early in the branch-and-bound procedure to reduce the search space.

A similar approach (based on the idea of consideration of the priorities of variables while branching) was investigated in [Pra87]. This research was aimed at improving the computational performance of the branch-and-bound solution procedure for the MILP model of register-transfer level logic design developed by Hafer [HP83]. In Hafer's model, essentially there are three types of binary variables. The first type (type I) decides which operator performs which operation, the second type (type II) decides which register stores which value, the third type (type III) decides whether a particular input value is accessed from a register or directly from the output of an operator. *A simple strategy of giving higher priority to type III variables over the type II variables lead to reducing the runtime to half its original value on the average.* This remarkable achievement of a simple strategy does make intuitive sense. Type III variables dictate whether or not a particular value should be stored (since if the stored copy of the value is never accessed, there is no need to keep a stored copy), whereas type II variables decide which register should be used for which value. It makes sense to first decide whether or not a stored copy is required and then to decide which register to use if any. Thus, the idea of assigning priorities to variables does have promise. Obviously, to assign priorities to the variables we require

problem-specific knowledge; i.e., use of knowledge about the problem to guide the branching process has promise for runtime improvement.

7.3.1.2 Generalized Design Strategy

Now, the question is how one goes about assigning priorities to the binary variables. The answer is that one has to know about the roles of the various variables in the problem being considered. Thus, in our MILP models, assigning priorities to binary variables is equivalent to assigning the same to design decisions. This is where design knowledge has to be used. As we said before, it may be effective to emulate an expert designer in assigning priorities. With this in mind, let us attempt to understand the design strategies used by human designers.

The question of how people perform system-level design is not very well understood. Usually, it is not a formalized process and involves a heuristic approach. The system is designed using a collection of independent rules and procedures which are applied on a case-by-case basis until the designer is satisfied that the design constraints have been met. The rules and procedures can be applied in different orders for different design problems. For a given problem, the order is not predetermined. It is determined dynamically in the design process. The next rule to be applied is dependent on how the design has progressed so far. Eventually, the designer stops when (s)he feels that all the design constraints have been met.

The point made by the above discussion is that the process of system-level design is really complex, and it is hard to exactly outline the way people perform such a design. One basic idea which is very relevant in this context is the idea of "focusing attention." The whole process of design seems to be revolving around this idea. In any reasonable-sized design, there are so many details involved that it is almost impossible for a human designer to keep track of the overall global view of the design at all times. Therefore, people try to break a bigger design into smaller designs. At a given time, they focus attention on a certain part of

the design. When it is done, the designer moves on to another part. The order in which various parts of the design are considered is decided by the designer based on his idea of the importance of various parts. The fact that the designer's idea of the importance of various parts is dynamic contributes to the complexity of the design strategy. The next part to be considered is dependent on how the design has progressed so far.

In the context of the application-specific multiprocessor system design problem considered in this thesis, for example, one scenario could be where a human designer picks up a part of the system which (s)he thinks is crucial to the performance of the overall system, then decides on some processors and communication links to be used in this part, then realizes that the rest of the decisions regarding this part could be made more efficiently if the design of another part of the system were known because of the interaction between the two parts. So (s)he shifts focus, designs the other part, comes back to finish the design of this part, then picks up some other less important part, and so on. So, the whole process or the strategy is very entangled in terms of the order of decisions. The general design strategy used by people being so complex, we consider some simplified strategies to assign priorities to binary variables in the next section.

7.3.1.3 Simplified Strategies

Now, we concentrate on the specific design problem considered in Section 3.2 and the corresponding MILP model. The MILP model consists of 7 different types of binary variables:

- $\sigma - type$
- $\gamma - type$
- $\delta - type$
- $\alpha - type$

- $\phi - type$
- $\beta - type$
- $\chi - type$

The generalized design strategy being so complex, we describe some simplified strategies to assign priorities. As mentioned in Section 7.3.1.1, two governing factors for motivation behind priority assignment are: (i) intuition, and (ii) constraint strengthening. We examine how each factor affects the priority assignment in the specific model under consideration.

Intuitive Ideas Essentially, there are two primary design aspects in the synthesis problem under consideration:

- Subtask computation: which processor executes a given subtask
- Data transfer: whether the transfer turns out to be remote or local

In the MILP model, $\sigma - type$ variables relate to the subtask computation design decisions, and $\gamma - type$ to the data transfer design decisions. Now, the question is which of the two types should be assigned a higher priority. In general, it may seem intuitive to assign higher priorities to $\sigma - type$ variables. Several other decisions become easier to make once relevant $\sigma - type$ variables have been fixed, for example, whether a particular data transfer should be of type remote or local ($\gamma - type$), and whether a particular processor is included in the system or not ($\beta - type$). Of course, the real issue is how one goes about fixing the $\sigma - type$ variables or what order to impose within the $\sigma - type$ variables. For a large design, it may be useful to assign different priorities to different $\sigma - type$ variables. One such possible ordering may be based on whether or not a particular $\sigma - type$ variable is related to the critical path of the design. If a subtask falls on the critical path of the design, then the $\sigma - type$ variables associated with this subtask may be assigned higher priority than the $\sigma - type$

variables associated with the subtasks not on the critical path. Of course, this is just one possibility among others. In any case, imposition of an ordering within the $\sigma - type$ variables would be an attempt to emulate the generalized design strategy discussed in the previous section and thus requires a good understanding of the same. We will not make such an attempt here and all the variables of the same type will be assigned the same priority in the simplified strategies.

Intuitively, $\sigma - type$ variables should be assigned a higher priority. But just to ensure the correctness of our intuition, we will also investigate the opposite case in our experiments. Thus, based on the two primary design issues, we consider two cases:

- $\sigma - type$ variables are assigned a higher priority than the $\gamma - type$ variables.
- $\gamma - type$ variables are assigned a higher priority than the $\sigma - type$ variables.

Constraint Strengthening In general, fixing any binary variable to a binary value leads to some degree of strengthening of the constraint system. Obviously, the question is which ones should be fixed first in order to achieve as much strengthening of the constraint system as possible and as early as possible during the branch-and-bound exploration. In this context, an important set of variables seems to be $\alpha - type$ variables in our MILP model. These variables dictate the order in which two subtasks are performed if they are both performed on the same processor, and thus ensure mutual exclusion. If we have already made the decision that a particular pair of subtasks is being executed by the same processor, then it is imperative that we fix the value of the corresponding $\alpha - type$ variable as soon as possible, and thus impose an ordering on the execution of the two subtasks. By allowing the $\alpha - type$ variable to continue to have a non-binary value in the current subproblem, we leave the constraint system very loose. The processor-usage-exclusion constraints (Equations 3.17 and 3.18) become effective only after the corresponding $\alpha - type$ variable has been fixed to a binary value. By letting the $\alpha - type$ variable have a non-binary value, the two subtasks may

continue to have their execution time intervals overlapping; and thus the task completion time may continue to look promising for the current subproblem; and we may continue to explore it. By fixing the $\alpha - type$ variable (and thus ensuring mutual exclusion), the task completion time may deteriorate; and as a matter of fact, we may even abandon the current subproblem because the completion time may violate the designer requirements or we may have already found another solution with a better completion time; and thus reduce the search time.

Basically, Equations 3.17 and 3.18 form a very loose system of constraints. One reason for this weakness is, as we just described, that the constraints just do not ensure the intended goal of mutual exclusion when necessary unless the corresponding $\alpha - type$ variable has been fixed to a binary value. There is another reason for the weakness and that has to do with the fact that the pair of constraints was obtained by linearizing the original processor-usage-exclusion constraint and with the way linearization was achieved. In general, linearization leads to introduction of some degree of looseness in the constraints. In this particular case, in fact, introduction of the large constant T_M could potentially introduce serious weakness in the constraint system. In order to get an insight into the weakness and the gravity of the problem, we examine an example scenario during the branch-and-bound exploration.

Let the value of constant T_M be $10,000$ (a value larger than the possible values of all the timing variables in the model). Let us say at some point in the exploration, subtasks S_{a1} and S_{a2} are mapped to the processor p_d; i.e, $\sigma_{d,a1} = \sigma_{d,a2} = 1$. So, Equations 3.17 and 3.18 have been effectively reduced to the following pair:

$$
\begin{aligned}
T_{SS}(S_{a2}) &\geq T_{SE}(S_{a1}) - (1 - \alpha_{a1,a2})10,000 \\
T_{SS}(S_{a1}) &\geq T_{SE}(S_{a2}) - (\alpha_{a1,a2})10,000
\end{aligned}
$$

Now, let us say at this point that $\alpha_{a1,a2} = 0$ represents an infeasible situation. This infeasibility may come about due to a given designer constraint that the

completion time for the overall task must not be greater than say 8000. This designer constraint coupled with other constraints reflecting the design problem may require that the subtask S_{a1} must be completed before time 7900. Also, let us say that due to other constraints in the design problem, the earliest subtask S_{a2} can be completed is at time 7900; and say the execution time of subtask S_{a1} on processor p_d is 100 time units. Now, consider $\alpha_{a1,a2} = 0$. The first of the above pair will be trivially satisfied in that case, and the second reduces to

$$T_{SS}(S_{a1}) \geq T_{SE}(S_{a2})$$

which implies that the subtask S_{a1} can not be completed before time 7900. Obviously, the only potential feasibility lies in the case $\alpha_{a1,a2} = 1$ now. Now, let us investigate what is the value to which $\alpha_{a1,a2}$ must be raised by the simplex algorithm in order to satisfy the given constraint scenario and thus make the current subproblem (where $\alpha_{a1,a2}$ has not yet been forced to a binary value) continue to look feasible. Well, the value is 0.01. At this value of $\alpha_{a1,a2}$, the first of the pair can still be trivially satisfied, and the second reduces to

$$T_{SS}(S_{a1}) \geq T_{SE}(S_{a2}) - 100$$

which implies that the subtask S_{a1} can be started at time 7800 and can thus be completed by time 7900. Obviously, mutual exclusion is not ensured in this situation (as both S_{a1} and S_{a2} are allowed to use processor p_d at the same time). The fact that such a small value of $\alpha_{a1,a2}$ has been able to satisfy the current constraint scenario, thus leading to further exploration of the current subproblem, poses a serious problem for the computational performance of the branch-and-bound procedure. Obviously, in the above situation, as we said before, the only hope lies in the case $\alpha_{a1,a2} = 1$ (of course, it is conceivable that even that case represents an infeasible situation). Ideally, we would like to be able to force $\alpha_{a1,a2} = 1$ right away. Even if that is not the case, we would like the constraint

system and the simplex algorithm to raise the value of $\alpha_{a1,a2}$ as close to 1 as possible. Because only when $\alpha_{a1,a2}$ is close to 1, there is a chance that we are realistically gauging the feasibility and/or potential of the current subproblem by looking at the corresponding LPR solution. However, the linearized constraints are so weak because of the large constant T_M that they get satisfied at such a low value of $\alpha_{a1,a2}$ in the above scenario. Of course, the primary culprit remains the branch-and-bound procedure which allows fractional values for binary variables during the exploration. One solution is obviously to force the variable to a binary value as soon as possible in the exploration. Thus, in order to strengthen the constraint system, it is crucial to fix the $\alpha - type$ variable in Equations 3.17 and 3.18 as soon as the $\sigma - type$ variables in those equations have been fixed to 1.

All the discussion above with respect to $\alpha - type$ variables is also applicable to $\phi - type$ variables. In fact, one could conclude that it is crucial to fix the $\phi - type$ variable in Equations 3.19 and 3.20 as soon as the $\sigma - type$ variables in those equations have been fixed to 1.

The Strategies Having had the discussion on intuition and constraint strengthening ideas, we are now ready to list the strategies with which we will experiment for priority assignment. As we said before, all the variables of the same type will be assigned the same priority. Intuitively, $\sigma - type$ variables should be assigned the highest priority. Then the discussion on constraint strengthening suggests that a given $\alpha - type$ variable should be fixed once the appropriate $\sigma - type$ variables have been fixed. Similarly, a given $\phi - type$ variable should be fixed after the appropriate $\sigma - type$ variables have been fixed. This suggests that $\sigma - type$ variables can be assigned priority level 1 (highest), $\alpha - type$ variables assigned level 2, and $\phi - type$ variables assigned level 3. All the rest of the binary variables can be assigned level 4. So, the most logical strategy for priority assignment is the ordering (σ, α, ϕ, rest). However, as we said before, we will also investigate the strategy where $\gamma - type$ variables are assigned a higher priority than the $\sigma - type$ variables. So, we experiment with the following two strategies:

- (σ, α, ϕ, rest) ordering
- (γ, σ, α, ϕ, rest) ordering

7.3.1.4 Experiments with the Simplified Strategies

In this section, we report experiments performed with the two strategies derived in the previous section. Once again, Bozo was used to solve the MILP models. However, the binary variables were assigned priorities in these experiments. There is an option in Bozo where such a specification of priorities is possible. Bozo will accept such priority assignment and use it in selection of branching variables.

Now, the question is how to measure the effectiveness of the strategies considered. Of course, one solution is to just compare the runtime with the case when no priority assignment was specified. However, to get a better analysis of the performance, we compare some other parameters also. As we know, the branch-and-bound exploration consists of solving several subproblems (solving each subproblem implies performing a number of simplex pivots). As the goal of incorporating intelligent branching strategies into the branch-and-bound procedure is to make the exploration more efficient, an effective strategy should reduce the number of subproblems explored and hence the total number of simplex pivots performed. Thus, in order to have an a detailed evaluation of the effectiveness of strategies, we compare the following three parameters:

- Bozo's runtime on a Solbourne Series5e/900 (similar to Sun SPARCsystem 4/490) with 128 MB of memory
- Total number of subproblems explored during the branch-and-bound
- Total number of simplex pivots performed in solving the subproblems

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	11.2	21	644
2	24.5	51	1285
3	26.5	61	1406
4	38.7	131	2170

Table 7.1: Runtime Statistics for No Priority Assignment (Four-Subtask Graph)

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	8.7	19	509
2	14.2	29	777
3	21.2	43	1153
4	12.3	37	670

Table 7.2: Runtime Statistics for (σ, α, ϕ, rest) Ordering (Four-Subtask Graph)

Strategy 1: (σ, α, ϕ, rest) Ordering Here, we used the four-subtask task graph example described in Section 4.1. Once again, 4 designs were generated using Bozo (priority assignment following the ordering was input to Bozo). Bozo's runtime, number of subproblems explored, and number of simplex pivots performed using the strategy are given in Table 7.2. To provide the comparison, Table 7.1 lists similar statistics when binary variables were not assigned any priorities.

A comparison of the data in Table 7.1 and Table 7.2 clearly shows that the strategy (σ, α, ϕ, rest) ordering leads to a significant and consistent improvement in the computational performance of the branch-and-bound. For each of the designs, each of the comparison parameters has been improved by the priority assignment. Bozo's runtime has been reduced to 56% of its original value on an average, the number of subproblems explored is reduced to 48%, and the number of simplex pivots to 56%.

Design

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	10.8	21	605
2	17.5	37	947
3	22.5	39	1168
4	50.8	171	2672

Table 7.3: Runtime Statistics for (γ, σ, α, ϕ, rest) Ordering (Four-Subtask Graph)

Strategy 2: (γ, σ, α, ϕ, rest) Ordering Once again, the four-subtask task graph example of Section 4.1 was used. Table 7.3 lists the runtime statistics for the priority assignment based on this strategy.

A comparison of the data in Table 7.1 and Table 7.3 shows that the strategy (γ, σ, α, ϕ, rest) ordering does not perform as well as the strategy (σ, α, ϕ, rest) ordering. In any case, this was intuitively expected. However, the strategy does lead to a runtime improvement in the first 3 designs (though to a lesser degree than the strategy 1). For the design 4, it leads to a significant deterioration in runtime.

Basically, strategy 1 seems to be much more effective and consistent. Thus, ordering (σ, α, ϕ, rest) seems to show a significant potential for improving runtime of branch-and-bound.

7.3.1.5 Dynamic Priorities

In the simplified strategies discussed in the previous two sections, we have only talked about the priorities of variables in a static sense; i.e., we assign relative priorities to the binary variables and then we run the branch-and-bound. Obviously, this static approach is not powerful enough to be able to emulate the generalized design strategy discussed in Section 7.3.1.2. As we saw, the generalized design strategy is very complex and dynamic in nature. In order to be able to emulate such a dynamic strategy, we have to make the priority of a variable a dynamic parameter during the branch-and-bound. Another motivating factor

for considering dynamic priorities is revealed from the discussion about the role of $\alpha - type$ and $\phi - type$ variables in constraint strengthening in Section 7.3.1.3. The discussion there implied that a particular $\alpha - type$ variable gains priority if we have already made the decision that the corresponding pair of subtasks is executed by the same processor. If the decision is that the two subtasks are executed by different processors, then it is not at all important to fix the $\alpha - type$ variable. Once again, to capture this scenario through priority assignment, the priority of a variable has to be a dynamic parameter.

As opposed to the static approach, the dynamic priority approach can change the priority of the variables at an intermediate stage during the branch-and-bound. The priority at a given stage is dependent on what decisions have already been made. It not only depends on what variables have already been fixed, but also on what binary values have been assigned to them in the current subproblem. In a sense, one has to look at the partial design constructed so far and then take the next decision. In this extreme case, the whole approach may be similar to what a human designer does; (s)he probably constructs some parts of the design and evaluates it and then decides what to do next. Eventually, the dynamic approach can take the form of an expert system having design knowledge built into it, which guides the branch-and-bound procedure.

The dynamic approach is worth investigating. Of course, apart from the simplified static strategies discussed in the previous two sections, several other static strategies are conceivable; and a significant amount of research should also be performed to investigate these other variations of static strategies. The knowledge about the effectiveness of various static strategies can be useful in developing a powerful dynamic approach.

7.3.2 Branching on a Group of Binary Variables

7.3.2.1 Motivation

The idea of assigning priorities to binary variables alone may not be powerful enough to simulate some sort of decision order used by a human designer. Many times, a single decision in the original design problem is encoded as a group of binary variables in the model, and so completing the decision in one branching would imply branching on the group of variables. Thus, it may be useful to explore a branching strategy which allows branching on a group of variables.

Another reason for group branching relates to the weakness introduced in the constraint system due to the linearization techniques used. As discussed in Section 7.3.1.3, linearization can render some constraints very weak and ineffective. To make these constraints effective in the system, at times, several binary variables must be fixed at the same time. To cite the example from Section 7.3.1.3, $\alpha - type$ variable should be fixed as soon as the $\sigma - type$ variables in Equations 3.17 and 3.18 have been fixed to 1.

Marsten and Morin [MM78] developed a technique known as *Resource Space Tour* (RST), which can be used to branch on several variables at once and efficiently evaluate the linear relaxations of the resulting subproblems to obtain upper bounds on their objective function values. The technique efficiently evaluates the whole set of resulting subproblems without evaluating each subproblem individually. Bozo uses this technique to reduce some computation.

Apart from possibly simulating a human designer decision, strengthening the constraint system, and reducing some computation by using RST, the strategy of branching on a group has one more advantage. Though each individual binary variable is allowed to have two values, often a group of variables is not allowed to have all possible vectors of values. As a simple example, consider the common case of a set of binary variables which form a specially ordered set of type 1 (i.e., the sum of the variables must equal 1). Exactly one variable is fixed to 1 in any feasible assignment of values, hence there are only n allowable vectors

of values instead of 2^n. So, by specifying the allowable vectors of values along with the branching group, we can prevent the creation of subproblems with values for the branching variables which do not make sense in the context of the design problem. The RST technique allows specification of such vectors and it only explores the allowed subproblems. As is obvious, this leads to a significant reduction in exploration. Another strength that comes from specification of branching groups and the corresponding vectors is as follows. As we know, one of the primary factors contributing to the weakness of the branch-and-bound procedure is the fact that it allows fractional values for binary variables during the exploration. We would like to be able to completely avoid such fractional value assignment if possible. Obviously that being difficult to achieve, we would like to minimize such cases. Some binary variables could be so closely linked to each other that if one is decided, the other can be predicted with almost certainty. In such a case, it makes no sense to leave the other variable hanging at a fractional value. One effective way to achieve this is by making so closely related variables part of a branching group and specifying the allowable vectors of values.

7.3.2.2 Possible Branching Groups

In the context of the MILP model of Section 3.2, some of the possible groups (along with the allowable vectors of values) worth exploring are (each group has been given a name for future reference)

- *σ-Group:* $\sigma_{d,a}$, for all d; i.e., all the $\sigma - type$ variables associated with a particular subtask S_a. Since each subtask is executed on exactly one processor, exactly one variable in the group is allowed to be 1 and others must be 0. So, there are n allowable vectors of values if there are n variables in the group.

- *δ, σ-Group:* $\delta_{d,a1,a2}, \sigma_{d,a1}, \sigma_{d,a2}$; i.e., a $\delta - type$ variable and its associated $\sigma - type$ variables. $\delta_{d,a1,a2}$ indicates whether or not the two subtasks S_{a1}

and S_{a2} are executed on the same processor p_d. $\delta_{d,a1,a2} = 1$ if and only if $\sigma_{d,a1} = \sigma_{d,a2} = 1$. So, there are only 4 allowable vectors of values for this group (instead of 8): $\{000, 001, 010, 111\}$.

- γ, δ*-Group:* $\gamma_{a1,a2}$ and $\delta_{d,a1,a2}$, for all d; i.e., a $\gamma - type$ variable and all the $\delta - type$ variables associated with it. In this group, either $\gamma_{a1,a2} = 1$ and all the other variables are zero, or $\gamma_{a1,a2} = 0$ and exactly one of the other variables is one. So, there are n allowable vectors of values if there are total n variables in the group.

- β, σ*-Group:* β_d and $\sigma_{d,a}$, for all a; i.e., a $\beta - type$ variable and all the $\sigma - type$ variables associated with it. Here, $\beta_d = 0$ if and only if all the other variables are zero. In all other cases, $\beta_d = 1$. It is easy to see that the number of allowable vectors of values is 2^{n-1} if n is the total number of variables in the group.

7.3.2.3 Experiments with Branching Groups

We performed experiments with two combinations of the branching groups described in the previous section. The first combination consists of σ-Group and γ, δ-Group, while the second consists of γ, δ-Group and β, σ-Group. The groups were specified to Bozo. Bozo accepts such group specification and uses the groups for branching.

Once again, to measure the effectiveness of the branching groups, we compare the runtime parameters with the case when no group branching was used. The runtime parameters considered are the same as in Section 7.3.1.4.

Group Combination 1: (σ-Group, γ, δ-Group) Once again, the four-subtask task graph example described in Section 4.1 was used. The groups were specified to Bozo, and 4 designs were generated as before for the example. Table 7.4 lists runtime statistics for this case.

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	15.6	26	866
2	20.1	47	1117
3	29.3	67	1712
4	39.5	149	2181

Table 7.4: Runtime Statistics for σ-Group, γ, δ-Group (Four-Subtask Graph)

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	12.1	37	791
2	19.5	49	1163
3	39.5	163	2727
4	33.9	179	1678

Table 7.5: Runtime Statistics for γ, δ-Group, β, σ-Group (Four-Subtask Graph)

We compare the data in Table 7.4 with that in Table 7.1 to evaluate the effectiveness of the group combination 1 specification for branching. Unfortunately, the comparison shows that the runtime has in fact deteriorated except for design 2.

Group Combination 2: (γ, δ-Group, β, σ-Group) Table 7.5 lists runtime statistics for this case for the four-subtask task graph example.

Once again, the comparison between the data in Table 7.5 and Table 7.1 does not lead to any definite conclusion. Runtime seems to have improved somewhat in designs 2 and 4, whereas it has deteriorated in designs 1 and 3.

It is difficult to say at this point which of the two group combinations, if any, is going to be effective in improving the branch-and-bound computational performance. In order to further investigate these group combinations and their effectiveness in branching, we combine these group combinations with priority assignment reflecting strategy 1 (σ, α, ϕ, rest) which had proved to be effective.

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	6.6	11	427
2	12.4	23	735
3	11.2	21	678
4	10.7	29	608

Table 7.6: Runtime Statistics for Group Combination 1 & (σ, α, ϕ, rest) Ordering (Four-Subtask Graph)

Group Combination 1 & (σ, α, ϕ, rest) Ordering Table 7.6 lists runtime statistics for the four-subtask task graph example when Bozo was run with group combination 1 specified for branching groups and binary variables assigned priorities based on (σ, α, ϕ, rest) ordering.

A comparison of the data in Table 7.6 and Table 7.1 shows a significant and consistent improvement in the runtime when this combination of group branching and priority assignment is used. For each of the designs, each of the comparison parameters has been improved by the combination of strategies. Bozo's runtime has been reduced to 40% of its original value on an average, number of subproblems explored is reduced to 32%, and the number of simplex pivots to 44%. In fact, the runtime performance has been improved with respect to the case when only the priority assignment for (σ, α, ϕ, rest) ordering was used (i.e., no group branching; Table 7.2). Bozo's runtime, number of subproblems, and number of simplex pivots have been reduced to 72%, 66%, and 79% respectively by addition of group branching to the priority assignment.

Group Combination 2 & (σ, α, ϕ, rest) Ordering Table 7.7 lists runtime statistics for the four-subtask task graph example when Bozo was run with group combination 2 specified for branching groups and binary variables assigned priorities based on (σ, α, ϕ, rest) ordering.

A comparison of the data in Table 7.7 and Table 7.1 shows that the group combination 2 added to the priority assignment does not perform as well as the

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	13.6	29	825
2	15.9	61	965
3	17.9	59	1152
4	8.3	31	490

Table 7.7: Runtime Statistics for Group Combination 2 & (σ, α, ϕ, rest) Ordering (Four-Subtask Graph)

group combination 1 added to the same. For design 1, in fact, this combination deteriorates the runtime even with respect to the case when no branching strategy (group or priority) was used to guide the branch-and-bound. Comparing Tables 7.7 and 7.2 shows no significant or consistent improvement in runtime by the addition of group combination 2 to the priority assignment. For designs 3 and 4, there seems to be some improvement; however, runtime has degraded for designs 1 and 2.

To summarize the experiments with the branching strategies so far, it seems that the use of group combination 1 (σ-Group, γ, δ-Group) coupled with the priority assignment based on (σ, α, ϕ, rest) ordering provides an effective and consistent branching strategy having a significant potential for improving the computational performance of the branch-and-bound exploration.

7.3.2.4 More Branching Groups

In Sections 7.3.2.2 and 7.3.2.3, we discussed some branching groups. In general, of course, there may be more such groups depending on the specific problem at hand, which is another research issue. In Section 7.3.2.1, one of the motivating factors to have branching groups was to be able to deal with the constraints rendered weak due to linearization. In this context, we cited the constraints represented by Equations 3.17 and 3.18. These weak linearized constraints make it desirable to have a branching group consisting of the $\alpha - type$ variable and

the $\sigma - type$ variables appearing in Equations 3.17 and 3.18. However, we never investigated this type of branching group later in our discussion.

The reason for not considering the group under question is that it is not very straightforward to deal with such a group. When we specify a branching group, we also need to specify possible and desirable vectors of values for the group. What are the desirable vectors for the group under question? The answer is not very clear. We know that corresponding $\alpha - type$ variable must be fixed to either 0 or 1 when both the $\sigma - type$ variables have been fixed to 1. In all other cases, we don't care about the value of the $\alpha - type$ variable; it could remain at 0, 1, or any intermediate value for that matter. So, it is not very clear and difficult to derive what the desirable vectors of values are for such a group.

How can we overcome the problem just described? In order to do so, we need to get an insight into the underlying cause of the problem. The root cause is that Equations 3.17 and 3.18 deal with two different design decisions in a complex manner. Basically, the two decisions are:

- Whether to serialize S_{a1} and S_{a2} or perform them in parallel?
- What processor is S_{a1} mapped to and what processor is S_{a2} mapped to?

The $\sigma - type$ variables clearly deal with the second decision. The $\alpha - type$ variable implicitly deals with the first decision. However, $\sigma - type$ variables also deal with the first decision in a complex way. Once a decision has been made that both S_{a1} and S_{a2} are mapped to the same processor (through $\sigma - type$ variables), it automatically implies that a decision has been made to serialize S_{a1} and S_{a2}. Once this decision has been made, then only the $\alpha - type$ variable comes into play and imposes an strict ordering between S_{a1} and S_{a2} for serialization. If S_{a1} and S_{a2} are mapped to different processors, then $\alpha - type$ variable plays no role and can continue to have any value. The fact that $\sigma - type$ variables deal with both the decisions in a very complex way leads to the situation that the value of $\alpha - type$ variable becomes redundant in many cases, thus making it difficult to derive the desirable vectors of values for the group. One solution

to this problem then is to reformulate the constraints where logically different decisions are clearly separated.

Indeed to derive more branching groups and to make them more effective, it may be desirable to reformulate some constraints at times. Particularly, reformulation where intuitively different decisions are clearly separated can be a very powerful tool. Such a reformulation will also lead to derivation of branching groups where each group represents a different design decision; and thus will make it easier to simulate the design strategy used by a human designer. Having discussed the usefulness of reformulation, we present a reformulation of the constraints represented by Equations 3.17 and 3.18, where each intuitive decision is represented by a different binary variable.

Reformulation of Processor-usage-exclusion Constraint Let us define a new binary variable $\rho_{a1,a2}$ to represent the serialization of subtasks S_{a1} and S_{a2}; i.e, $\rho_{a1,a2} = 1$ implies S_{a1} and S_{a2} are to be serialized. Let $\alpha_{a1,a2}$ in this reformulation represent that S_{a1} is performed first in the case of serialization (similarly, $\alpha_{a2,a1}$ represents that S_{a2} is performed first in the case of serialization). Now, Equations 3.17 and 3.18 can be rewritten as the set of following four relations:

$$T_{SS}(S_{a2}) \geq T_{SE}(S_{a1}) - (1 - \alpha_{a1,a2})T_M \tag{7.1}$$

$$T_{SS}(S_{a1}) \geq T_{SE}(S_{a2}) - (1 - \alpha_{a2,a1})T_M \tag{7.2}$$

$$\alpha_{a1,a2} + \alpha_{a2,a1} = \rho_{a1,a2} \tag{7.3}$$

$$\sigma_{d,a1} + \sigma_{d,a2} \leq \rho_{a1,a2} + 1 \tag{7.4}$$

In the above reformulation, one possible branching group is ($\alpha_{a1,a2}$, $\alpha_{a2,a1}$, $\rho_{a1,a2}$). It is easy to see that there are only 3 allowable vectors of values for this group (instead of 8): $\{000, 011, 101\}$.

7.4 Constraint Satisfaction

In this section, we discuss one more aspect of the LP-based branch-and-bound which needs to be strengthened to improve its computational performance. An LP-based branch-and-bound method for solving MILP essentially searches for a better MILP solution that improves the objective function further. In its quest for a better solution, at each step it forces certain binary variables to certain binary values and checks to see if the set of constraints is still satisfied; whenever it finds the constraints are not satisfied, it abandons the current search and attempts another search direction. Now, the question is how it checks if the constraints are still satisfied. In the branch-and-bound steps mentioned in Section 7.2, none of the steps explicitly mention any check for constraint satisfaction. Well, actually the check is embedded in the LPR solution step. However, this does not seem to be the natural way to check constraint satisfaction. In an LPR solution, the binary variables which are not yet forced to a specific binary value are allowed to have a fractional value. Because of this flexibility in the LPR solution, it may seem that the constraints are still satisfiable; whereas if all the binary variables were forced to binary values, it might have been impossible to satisfy the constraints. So, a subproblem which has actually become infeasible keeps getting explored further because of this "flexible" constraint satisfaction check. Early detection of such cases will improve computational performance. In fact, we have elaborated on this problem of loose constraints in quite detail in Section 7.3.1.3, where we discussed how Equations 3.17 and 3.18 seem falsely satisfiable when the $\alpha - type$ variable is allowed to have a fractional value. Here, we outline some techniques that can be directly brought to bear upon the constraint satisfaction aspect of branch-and-bound.

7.4.1 Constraint Satisfaction Techniques

There have been some efficient AI techniques developed for constraint satisfaction problems [Nad86]. A survey of these techniques is available in [Nad89]. Essentially, the constraint satisfaction problems consist of variables, values and constraints; and the goal is to find assignment of the variables to their candidate values such that all the constraints are satisfied. In our MILP model, we have several variables and constraints, and we wish to assign values to the variables such that constraints are satisfied (and of course, in our case we also wish to find values such that a given objective function is optimized). So, it seems that the constraint satisfaction techniques can be useful in dealing with this aspect of the MILP model. In fact, Sidebottom [Sid90] has developed some powerful techniques for constraint satisfaction which can be applicable to MILP problems. Here, we give a quick overview of Sidebottom's technique.

Essentially, Sidebottom assumes that a set of variables is given with a range of possible values for each variable. Also, a set of constraints is given. The goal is to find feasible value assignments to the variables such that the set of constraints is satisfied. Basically, the technique uses an iterative improvement approach. It starts with an upper and lower bound for each variable (which can be derived from the given ranges of variables). Then during each iteration, it tries to reduce the range of possible values for variables by improving lower and upper bounds of the variables. It does so by looking at the constraints and deriving the upper and lower bounds on the left hand side and the right hand side of the constraints, and then attempting to refine these bounds which eventually lead to refinement of the bounds on the variables. If the upper/lower bounds for the variables can not be improved any further, then we know that the feasible range of values for each variable has been found (which satisfies the set of constraints). We illustrate the technique more concretely with an example of how it can be applied to constrain the values of variables in our MILP model. In the following, the notation $\downarrow (x)$ represents a lower bound on x, and $\uparrow (x)$ an upper bound on x.

Let us use the same example scenario to illustrate the technique that we used to illustrate the weakness of the linearized constraints in Section 7.3.1.3. There we looked at a point during the branch-and-bound exploration when subtasks S_{a1} and S_{a2} have been mapped to the processor p_d. So, the constraints under consideration are (in this technique as opposed to MILP solution techniques, the constraints do not have to be in a linear form; so we consider the following non-linear form of the constraints):

$$T_{SS}(S_{a2}) \geq \alpha_{a1,a2} T_{SE}(S_{a1}) \tag{7.5}$$

$$T_{SS}(S_{a1}) \geq (1-\alpha_{a1,a2}) T_{SE}(S_{a2}) \tag{7.6}$$

Now, assuming the same constraint scenario as described in Section 7.3.1.3, we have the following bounds available on the variables:

$$\downarrow (T_{SS}(S_{a2}) = 0; \uparrow (T_{SS}(S_{a2}) = 10,000$$

$$\downarrow (T_{SE}(S_{a1}) = 0; \uparrow (T_{SE}(S_{a1}) = 7900$$

$$\downarrow (T_{SS}(S_{a1}) = 0; \uparrow (T_{SS}(S_{a1}) = 7800$$

$$\downarrow (T_{SE}(S_{a2}) = 7900; \uparrow (T_{SE}(S_{a2}) = 10,000$$

$$\downarrow (\alpha_{a1,a2}) = 0; \uparrow (\alpha_{a1,a2}) = 1$$

Let us represent the left hand side of Equation 7.5 by $C1L$ and right hand side by $C1R$, and those of Equation 7.6 by $C2L$ and $C2R$ respectively. Then using the bounds available on the variables, we can come up with the following bounds: $\downarrow (C1L) = 0$; $\uparrow (C1L) = 10,000$; $\downarrow (C1R) = 0$; $\uparrow (C1R) = 7900$; $\downarrow (C2L) = 0$; $\uparrow (C2L) = 7800$; $\downarrow (C2R) = 0$; $\uparrow (C2R) = 10,000$. Now, a comparison of $\uparrow (C2L)$ and $\uparrow (C2R)$ clearly shows that $\uparrow (C2R)$ can be refined to be 7800. By expanding $C2R$ to its original expression, this new bound implies $\uparrow (1-\alpha_{a1,a2}) = 78/79$ which eventually implies $\downarrow (\alpha_{a1,a2}) = 1/79$. So, by using

the original bounds given on the variables, the technique has been able to refine the lower bound on $\alpha_{a1,a2}$ to be 1/79; which obviously implies that $\alpha_{a1,a2}$ being a binary variable must be fixed to 1. Since the technique will have the knowledge that $\alpha_{a1,a2}$ is a binary variable, it can easily flag that it be fixed to 1. As the reader would recall from the discussion in Section 7.3.1.3, the value of $\alpha_{a1,a2}$ remained at 0.01 in the original branch-and-bound, though we would have liked to force it to 1 immediately. Use of the constraint satisfaction technique can help in forcing the variable to 1, as we just illustrated.

Thus, it seems that Sidebottom's technique can be used in conjunction with branch-and-bound. The technique can be used dynamically during the branch-and-bound. At any point during the branch-and-bound, certain binary variables would have been forced to specific binary values, and the technique can be used to refine the upper and lower bounds on the variables based on the new forced values to variables. Any refinement of bounds on binary variables immediately forces them to a specific value. Of course, the technique also provides useful bounds on non-binary variables as well. These bounds can be helpful in fathoming subproblems also. Intuitively, the technique can be expected to significantly reduce the exploration during the branch-and-bound.

Sidebottom's technique has indeed been incorporated into the branch-and-bound procedure of Bozo and a new program called *Bonsai* has been developed by Hafer [Haf91]. This new program was used to perform some experiments to see how effective the constraint satisfaction technique is in improving the performance of branch-and-bound.

7.4.2 Experiments and Results

7.4.2.1 Bonsai Alone

Once again, the four-subtask task graph example described in Section 4.1 was used. Bonsai was used instead of Bozo to investigate the effectiveness of the

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	17.0	31	728
2	25.8	51	1094
3	54.5	109	2481
4	12.9	29	499

Table 7.8: Bonsai Runtime Statistics (Four-Subtask Graph)

constraint satisfaction technique. 4 designs were generated as before. Table 7.8 lists runtime statistics for this case.

We compare the data in Table 7.8 with that in Table 7.1 to evaluate the effectiveness of Bonsai. Unfortunately, the comparison shows that the runtime has in fact deteriorated except for design 4. However, a closer observation reveals that the number of subproblems and the number of simplex pivots have been reduced on an average, though the runtime has increased. In fact, Number of subproblems and number of simplex pivots have been reduced to 83% and 87% respectively, while the runtime is increased to 109%. It seems that the constraint satisfaction was able to fathom several subproblems which led to reduction in the two numbers; however, running of constraint satisfaction itself required significant computer time which led to an overall increase in the runtime.

So, it is difficult to draw any definite conclusions about the constraint satisfaction technique at this point. To further investigate the matter, we combine Bonsai with some of the branching strategies.

7.4.2.2 Bonsai Combined with (σ, α, ϕ, rest) Ordering

Table 7.9 lists runtime statistics for the four-subtask task graph example when Bonsai was run with priority assignment based on (σ, α, ϕ, rest) ordering.

A comparison of the data in Table 7.9 and Table 7.1 shows a significant and consistent improvement in the runtime now. For each of the designs, each of the comparison parameters has been improved by the combination of constraint satisfaction and (σ, α, ϕ, rest) ordering. Bozo's runtime has been reduced to 48%

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	10.0	17	464
2	15.5	29	689
3	15.6	25	693
4	7.6	15	345

Table 7.9: Bonsai Runtime Statistics for (σ, α, ϕ, rest) Ordering (Four-Subtask Graph)

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	14.2	17	619
2	19.7	34	989
3	18.5	28	945
4	18.3	54	642

Table 7.10: Bonsai Runtime Statistics for σ-Group, γ, δ-Group (Four-Subtask Graph)

of its original value on an average, number of subproblems explored is reduced to 32%, and the number of simplex pivots to 40%. Comparison of performance with the case when only the priority assignment for (σ, α, ϕ, rest) ordering was used does not show such consistency. Runtime has improved for designs 3 and 4 while it has deteriorated for designs 1 and 2. However, the average performance has definitely improved. In fact, Bozo's runtime, number of subproblems, and number of simplex pivots have been reduced to 86%, 67%, and 70% respectively by addition of constraint satisfaction to the priority assignment.

7.4.2.3 Bonsai Combined with Group Combination 1

Table 7.10 lists runtime statistics for the four-subtask task graph example when Bonsai was run with group branching for group combination 1.

A comparison of the data in Table 7.10 and Table 7.1 shows an improvement in runtime except for design 1. Bozo's runtime has been reduced to 70% of its original value on an average, the number of subproblems is reduced to 50%, and

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	14.1	25	641
2	20.0	36	981
3	23.4	60	1134
4	23.9	74	952

Table 7.11: Bonsai Runtime Statistics for γ, δ-Group, β, σ-Group (Four-Subtask Graph)

the number of simplex pivots to 58%. As we recall, group combination 1 by itself had not performed well at all. Addition of constraint satisfaction to the strategy has really improved the performance.

7.4.2.4 Bonsai Combined with Group Combination 2

Table 7.11 lists runtime statistics for the four-subtask task graph example when Bonsai was run with group branching for group combination 2.

A comparison of the data in Table 7.11 and Table 7.1 shows an improvement in runtime except for design 1. Bozo's runtime has been reduced to 80% of its original value on an average, number of subproblems is reduced to 74%, and the number of simplex pivots to 67%. As we recall, group combination 2 by itself also had not performed well at all. Addition of constraint satisfaction to the strategy has once again improved the performance.

7.4.2.5 Bonsai Combined with Group Combination 1 & (σ, α, ϕ, rest) Ordering

Table 7.12 lists runtime statistics for the four-subtask task graph example when Bonsai was run with group combination 1 specified for branching groups and binary variables assigned priorities based on (σ, α, ϕ, rest) ordering.

A comparison of the data in Table 7.12 and Table 7.1 shows a significant and consistent improvement in the runtime by the combination of constraint satisfaction, group combination 1, and (σ, α, ϕ, rest) ordering. For each of the

Design	Runtime (sec)	# of Subproblems	# of Simplex Pivots
1	8.1	11	412
2	13.3	21	624
3	11.0	17	543
4	6.9	14	358

Table 7.12: Bonsai Runtime Statistics for Group Combination 1 & (σ, α, ϕ, rest) Ordering (Four-Subtask Graph)

designs, each of the comparison parameters has been improved. Bozo's runtime has been reduced to 39% of its original value on an average, number of subproblems is reduced to 24%, and the number of simplex pivots to 35%. Comparison of performance with the case when Bozo was used with group combination 1 and (σ, α, ϕ, rest) ordering does not show such consistency. Runtime has improved for designs 3 and 4 while it has deteriorated for designs 1 and 2. However, the average performance has somewhat improved. Bozo's runtime, number of subproblems, and the number of simplex pivots have been reduced to 96%, 75%, and 79% respectively by addition of constraint satisfaction to group branching and priority assignment.

By looking at all the experiments performed using Bonsai, it seems that the use of a constraint satisfaction technique in conjunction with group combination 1 (σ-Group, γ, δ-Group) and priority assignment based on (σ, α, ϕ, rest) ordering provides an effective strategy, and has a significant potential for improving the runtime. With this conjecture in mind, we performed some experiments with the nine-subtask task graph.

7.5 Some Experiments with Nine-Subtask Task Graph

In this section, we report some experimental results for runtime improvement using the nine-subtask task graph example described in Section 4.4. First, we

Design	Runtime (min)	# of Subproblems	# of Simplex Pivots
1	62.2	603	30555
2	445.17	4931	198670
3	538.67	5265	281385
4	75.18	453	42546
5	6416.87	95257	3196180

Table 7.13: Runtime Statistics for No Branching Strategy (Nine-Subtask Graph)

Design	Runtime (min)	# of Subproblems	# of Simplex Pivots
1	35.62	287	18136
2	519.51	4029	266948
3	138.55	1103	71759
4	22.78	139	11846
5	546.2	7393	274820

Table 7.14: Runtime Statistics for (σ, α, ϕ, rest) Ordering (Nine-Subtask Graph)

experimented with the simple branching strategy of priority assignment for (σ, α, ϕ, rest) ordering. This was done to get a feel for what kind of runtime improvements can be expected for this example.

7.5.1 (σ, α, ϕ, rest) Ordering

Once again, 5 designs were generated using Bozo with priority assignment based on (σ, α, ϕ, rest) ordering. Table 7.14 lists the runtime statistics. To provide the comparison, Table 7.13 lists similar statistics when Bozo was used with no branching strategy specified.

A comparison of the data in Table 7.13 and Table 7.14 clearly shows that the strategy (σ, α, ϕ, rest) ordering leads to a significant improvement in the computational performance for nine-subtask task graph example (except for design 2). Bozo's runtime has been reduced to 17% of its original value on an average, the number of subproblems is reduced to 12%, and the number of simplex pivots to 17%. For this simple strategy, it is indeed a very significant improvement in

Design	Runtime (min)	# of Subproblems	# of Simplex Pivots
1	16.22	252	6989
2	76.23	1124	42713
3	122.44	1445	66998
4	7.53	85	3304
5	124.39	2500	29051

Table 7.15: Bonsai Runtime Statistics for Group Combination 1 & (σ, α, ϕ, rest) Ordering (Nine-Subtask Graph)

runtime. Runtime is improved by almost an order of magnitude. It seems that the strategies discussed in this chapter will show better improvement as we go to larger examples. Next, we run Bonsai with group combination 1 and (σ, α, ϕ, rest) ordering for this example.

7.5.2 Bonsai with Group Combination 1 & (σ, α, ϕ, rest) Ordering

Here, we experiment with the strategy that displayed maximum potential with the four-subtask task graph example. Once again, 5 designs were generated using Bonsai with branching groups and priority assignment specified. Table 7.15 lists the runtime statistics.

A comparison of the data in Table 7.15 and Table 7.14 clearly shows that the strategy used leads to an enormous and consistent improvement in the computational performance for the nine-subtask task graph example. For each of the designs, each of the comparison parameters has been improved. Bozo's runtime has been reduced to 5% of its original value on the average, the number of subproblems is reduced to 5%, and the number of simplex pivots to 4%. Runtime is improved by almost two orders of magnitude.

7.5.3 Discussion

Experiments performed with nine-subtask task graph example lead to a strong belief that the strategies discussed in this chapter will become more and more useful as we go to larger examples. Runtime improvements demonstrated in this chapter are very significant. Definitely, out of all the strategies discussed, the use of a constraint satisfaction technique in conjunction with group combination 1 and (σ, α, ϕ, rest) ordering provides the most powerful strategy.

This chapter has clearly showed that there is hope for significant runtime improvement. Of course, more research along these lines is needed. We need to investigate further the strategies discussed here and also discover more such strategies.

Chapter 8

Development of Some Theory and Useful Bounds

In this chapter, we attempt to develop a theoretical basis for the synthesis problem under consideration. Development of such a basis would be useful in three ways:

- It would lead to a deeper understanding of the synthesis problem.
- It can be used to generate tighter bounds which would be useful in reducing the search space for the problem.
- Theoretical results can be used as a basis for the development of algorithmic/heuristic procedures for synthesis.

The prime motivating force in this work is to have an ability to compute useful bounds on various design parameters of interest.

8.1 Motivation for Computing Bounds

The knowledge about the bounds on the various parameters of the design is usually very helpful in the design process. In our case, such bounds would be useful in two ways. They could be used to improve the generated MILP model itself. If we can come up with certain bounds, a tighter model can be generated to

reflect the reduced search space. For example, if an upper bound on the number of processors to be included can be found, variables and constraints in the model can be tailored to reflect this information. A tighter model (which translates into smaller search space) will likely lead to a smaller runtime. Bounds can also be used to guide the branch-and-bound procedure and thus reduce the runtime. For example, if the objective function value associated with a subproblem is beyond the bounds, there is no need to further explore the subproblem. Also, given a lower bound on the objective function value (for a MIN problem), if we happen to stumble upon a solution with objective function value equal to the bound during the branch-and-bound, there is no need to continue the exploration any further since no better solution could be found. Of course, if the optimality of the solution is not too crucial, then as soon as a solution with objective function value close to the bound (how close, specified by designer) is detected, we can stop. Thus, bounds can be useful in several ways in solving a design problem. In the ADAM system [PHJ+88, JKMP89], bounds have been effectively used to predict the design space as well as to explore it efficiently for datapath synthesis [JMP88, JPP87, PPM86, PP86].

Bounds for the synthesis problem under consideration have not been investigated. As we mentioned in Section 2.1.4, some bounds for scheduling precedence graphs on homogeneous systems have been reported in [FB73, AM90]. However, these bounds are approximate. Fernandez and Bussell [FB73] completely ignore communication, and present bounds to achieve the minimum execution time. Al-Mouhamed [AM90] presents a lower bound on the number of processors to achieve the minimum execution time, however, he does not determine an upper bound. These bounds are not very useful for the heterogeneous system problem we are considering. In our case, the cost as well as the performance of the system is considered. We need to compute lower and upper bounds on cost, performance, number of processors, and number of communication links. Let us examine how we can compute some bounds.

8.2 Bounds on Cost and Performance

Let us consider a slightly simplified version of the problem described in Section 3.2. We assume that $f_R(i_{a,b})$ and $f_A(o_{a,c})$ parameters are zero for all the inputs and outputs; i.e., a subtask requires all the inputs before it can start and all the outputs from the subtask become available only after the subtask is completed. So, what we have is a given set of subtasks $T = \{S_1, S_2, ..., S_n\}$, and a given set of processor types[8.1] $P_t = \{t_1, t_2, ..., t_k\}$. For each subtask S_a, we have a set of processor types $P_{t,a} \subseteq P_t$, which represents the types of processors capable of executing it. A parameter[8.2] $D_{tS}(t_i, S_a)$ represents the execution time for the subtask S_a when performed on a processor of type t_i. There are edges in the task graph which represent the dependencies and data transfers among the subtasks in the set T. Let us say there are m such edges and they are represented by the set $D = \{d_1, d_2, ..., d_m\}$. Each d_i can be expressed in the form (S_{i1}, S_{i2}), and it implies a data transfer from subtask S_{i1} to subtask S_{i2}. A parameter V_i represents the volume of data transfer for the edge d_i. Another parameter D_{CR} represents the time taken in transferring a unit volume of data over a communication link. Each of the processor types t_i has a cost associated with it C_i. The cost associated with the creation of a communication link is C_L.

8.2.1 Bounds on the Number of Processors

One of the interesting parameters for the design is the required number of processors of a given type. An upper bound on the number of processors of a given type t_i can be estimated by analyzing the maximum possible parallelism in the task graph. We need to look at a reduced task graph which consists of only the subtasks that are executable by the processor type t_i; i.e, all the subtasks in the

[8.1]Please note the notation is slightly different here; P_t represents the set of *processor types*, not processors.

[8.2]The parameter D_{tS} represents the execution time for the subtask S_a when a *type-t_i* processor is used.

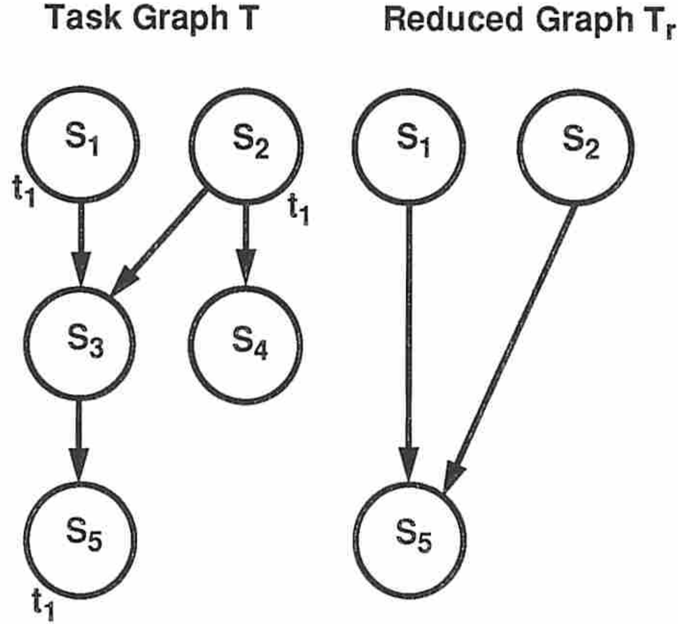


Figure 8.1: Example for Upper Bound on Number of Processors

set $T_i = \{S_a | t_i \in P_{t,a}\}$. The reduced task graph will represent the dependencies among the subtasks in the set T_i. Pictorially, the reduced graph is obtained from the original graph by deleting the subtasks not in T_i and for each deleted subtask connecting each of its inputs to each of its outputs. Now, upper bound on the number of processors of type t_i will be obtained by analyzing the maximum possible parallelism in the reduced graph; i.e, we find the largest set of subtasks ($T_{p,i} \subseteq T_i$) in the graph such that all the subtasks in $T_{p,i}$ can be performed independent of each other or in other words, in parallel. The cardinality of the set $T_{p,i}$ gives an upper bound on the number of processors of type t_i. This is illustrated by an example in Figure 8.1. The task graph T consists of 5 subtasks S_1, S_2, S_3, S_4, and S_5. Only 3 of them (S_1, S_2, and S_5) are executable by a type t_1 processor. The reduced graph T_r is as shown in the figure. Now, the largest set of subtasks executable in parallel in the reduced graph is $\{S_1, S_2\}$. So, an upper bound on the number of processors of type t_1 is 2.

A lower bound on the number of processors of a given type t_i is essentially zero, unless there is a subtask which can only be performed by the type under consideration. So, the lower bound is unity if there exists an S_a such that $P_{t,a} = \{t_i\}$ and it is zero otherwise.

8.2.2 Bounds on the Number of Communication Links

This is another useful parameter. An upper bound on the number of communication links is m, where m is the total number of data transfers in the task graph. The lower bound is obviously zero because a uniprocessor system may be synthesized. However, if there are data transfers $d_i = (S_{i1}, S_{i2})$ such that $P_{t,i1} \bigcap P_{t,i2} = \phi$ (i.e., the subtasks S_{i1} and S_{i2} can never be executed on the same processor), then a uniprocessor system is never feasible. So, if the set $D_m = \{d_i | P_{t,i1} \bigcap P_{t,i2} = \phi\}$ is a non-empty set, then a lower bound on the number of communication links is one.

8.2.3 Bounds on the Cost of the Synthesized System

The cost of the system is certainly an important parameter. An upper bound on the cost of the system is given by the sum of the costs of the processors having largest costs for each of the subtasks and the cost associated with creation of m communication links. More precisely, the upper bound is given by

$$C_{ub} = mC_L + \sum_{a|S_a \in T} \max_{i|t_i \in P_{t,a}} C_i \tag{8.1}$$

A lower bound on the cost, assuming a uniprocessor configuration, is given by the cost of the least expensive processor, or more precisely

$$C_{lb} = \min_{i|t_i \in P_t} C_i \tag{8.2}$$

8.2.4 Bounds on the Performance of the System

The performance of the system (represented by the task completion time) is another crucial parameter. An upper bound on the task completion time (TC) is given by the sum of the execution times of each of the subtasks when each subtask is performed on the slowest processor and the delays associated with each of the data transfers assuming each data transfer edge translates into a data transfer over a communication link. More precisely,

$$TC_{ub} = \Big(\sum_{a|S_a \in T} \max_{i|t_i \in P_{t,a}} D_{tS}(t_i, S_a)\Big) + \Big(D_{CR} \sum_{i|d_i \in D} V_i\Big) \tag{8.3}$$

A lower bound on the task completion time can be obtained by computing the critical path delay for the task graph. Each node (subtask S_a) in the graph is assigned a delay which is the fastest possible execution time for that node; i.e., node S_a is assigned a delay of $\min_{i|t_i \in P_{t,a}} D_{tS}(t_i, S_a)$. However, there is one more complication since the edges in the graph also have a potential of contributing to the task completion time delay. For the purpose of obtaining a lower bound, delays due to edges can be ignored except for the edges that belong to the set D_m defined above in the context of computing bounds on the number of communication links. An edge $d_i \in D_m$ would definitely contribute a delay of $V_i D_{CR}$ in any synthesized system. These delays can be accounted for while computing the critical path delay.

As the reader would have noticed, the bounds presented in this section are not very tight bounds. In Section 8.3, we make an attempt to estimate some tighter bounds and discuss some of the associated difficulties and problems.

8.3 Tighter Bounds: Associated Difficulties and Problems

Two of the factors contributing to the difficulties in finding tighter bounds for the synthesis problem at hand are as follows:

- We are considering a heterogeneous system. A given subtask may be executed on any of several processor types. Thus, the delay associated with a subtask is not known a priori.
- Interprocessor communication is an important tradeoff parameter, and the interconnection delay is crucial. A pair of communicating subtasks may be mapped to the same processor or different processors. So, an edge connecting two subtasks may or may not lead to an interconnection delay, and there is no a priori knowledge about it.

Let us look at some of the complications in computing bounds.

8.3.1 Critical Path Estimation

To compute the critical path (the path that leads to the largest delay) for the task graph, we have to know the delays associated with the subtasks and the data transfers of the graph. This path determines the minimum possible completion time for the task. To get a lower bound on the critical path delay, one can assign the fastest possible execution time for each node and ignore the delays associated with most of the edges, except for the edges in the set D_m, as we proposed in Section 8.2. However, ignoring the communication delays may not give a very tight bound.

One is tempted to suggest the most intuitive strategy to improve the quality of bound. For a given edge $d_i = (S_{i1}, S_{i2})$, if the fastest processor type for S_{i1} is different from the fastest processor type for S_{i2}, then we attach a communication

delay of $V_i D_{CR}$ with the edge. Thus, we would have accounted for some of the communication delays and could obtain an improved lower bound on the critical path delay. Now, the question is whether this strategy would give a correct lower bound, and the answer unfortunately is *not in all cases.*

Before we look at examples illustrating the problem with the strategy and go into a detailed discussion of complications in estimating critical path, let us point out that the major difficulty in estimation is caused by the interprocessor communication factor. It turns out that the presence of heavy inter-subtask communication makes the estimation process hard. However, the initial partitioning of a task into component subtasks is usually performed with one of the goals being to keep the inter-subtask communication low. For such "well-partitioned" tasks, our estimation of critical path will likely provide useful bounds for guidance of branch-and-bound. With this in mind, we look at the problems with critical path estimation next.

Consider a small task graph shown in Figure 8.2. There are two subtasks S_{i1} and S_{i2} and a data transfer $d_i = (S_{i1}, S_{i2})$. Let us say we have three types of processors t_1, t_2, and t_3. The execution times for S_{i1} are 5, 15 and 7 time units on processor types t_1, t_2 and t_3 respectively. The corresponding numbers for S_{i2} are 14, 5 and 8 respectively. The volume of data at d_i is 8 units, and $D_{CR} = 1$. Now, if we followed the above strategy of assigning the fastest processors to the subtasks and creating a link if an edge connects two different processor types, we would assign a delay of 5 (processor type t_1) to S_{i1}, a delay of 5 (processor type t_2) to S_{i2} and a delay of 8 to the edge d_i. The lower bound thus obtained is $5 + 5 + 8 = 18$ time units. However, it is obvious that a lower delay of $7 + 8 = 15$ time units is achievable by assigning both S_{i1} and S_{i2} to a t_3 type processor.

The example discussed above reveals one counter-intuitive result; i.e, executing each individual subtask on a processor type leading to the fastest execution time for the subtask does not necessarily lead to the fastest completion time for the overall task. The following theorem states this result more precisely.

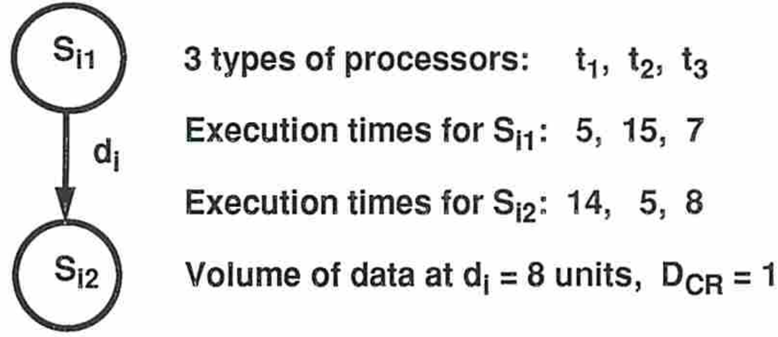


Figure 8.2: An Example to Illustrate a Problem with Critical Path Estimation

Theorem 8.3.1.1 *Minimization of the task completion time does not require the individual subtasks to be mapped to the fastest processor types; i.e., the mapping does not have to satisfy the condition that a given subtask $S_a \in T$ is mapped to a processor type t_k such that $D_{tS}(t_k, S_a) = \min_{i|t_i \in P_{t,a}} D_{tS}(t_i, S_a)$.*

Proof: The proof is performed using a hypothetical example. Consider the graph in Figure 8.2. Let t_{k1} and t_{k2} be the fastest processor types for S_{i1} and S_{i2} respectively and $k1 \neq k2$; i.e.,

$$D_{tS}(t_{k1}, S_{i1}) = \min_{i|t_i \in P_{t,i1}} D_{tS}(t_i, S_{i1})$$
$$D_{tS}(t_{k2}, S_{i2}) = \min_{i|t_i \in P_{t,i2}} D_{tS}(t_i, S_{i2})$$

Now, let us say, t_c is a processor type capable of executing both S_{i1} and S_{i2}, and the cumulative execution time for the pair of subtasks, when allocated to a single processor, is minimum on a processor of type t_c; i.e.,

$$D_{tS}(t_c, S_{i1}) + D_{tS}(t_c, S_{i2}) = \min_{i|t_i \in P_{t,i1} \cap P_{t,i2}} (D_{tS}(t_i, S_{i1}) + D_{tS}(t_i, S_{i2}))$$

Obviously,

$$D_{tS}(t_{k1}, S_{i1}) + D_{tS}(t_{k2}, S_{i2}) \leq D_{tS}(t_c, S_{i1}) + D_{tS}(t_c, S_{i2})$$

Now, let the interprocessor communication delay ($= V_i D_{CR}$) due to edge d_i satisfy the following condition:

$$V_i D_{CR} > (D_{tS}(t_c, S_{i1}) + D_{tS}(t_c, S_{i2})) - (D_{tS}(t_{k1}, S_{i1}) + D_{tS}(t_{k2}, S_{i2}))$$

It is easy to see that the task completion time $(D_{tS}(t_{k1}, S_{i1}) + D_{tS}(t_{k2}, S_{i2}) + V_i D_{CR})$ is not minimized when the individual subtasks are mapped to the fastest processor types. Q.E.D.

The above proof illustrates that the simple strategy of mapping the subtasks to the fastest processor types does not optimize the overall performance because of the presence of interprocessor communication delays. As a matter of fact, the role played by the interprocessor communication can be expressed concretely by the following corollary.

Corollary 8.3.1.1 *For a pair of subtasks S_{i1} and S_{i2} connected by an edge $d_i = (S_{i1}, S_{i2})$; if t_{k1} and t_{k2} are the fastest processor types for S_{i1} and S_{i2} respectively and $k1 \neq k2$, and t_c is the processor type (among all types capable of executing both S_{i1} and S_{i2}) for which the sum of the individual execution times is minimum, then the execution of S_{i1} and S_{i2} on the fastest processor types minimizes the overall completion time for the pair if and only if the interprocessor communication delay contributed by the edge d_i is not greater than the difference between the sum of the execution times of both S_{i1} and S_{i2} on the type t_c and the sum of the execution times of S_{i1} and S_{i2} on the types t_{k1} and t_{k2} respectively.*

Another problem faced in critical path estimation is illustrated with the help of the graph shown in Figure 8.3. Assume t_k is the fastest processor type for each of the three subtasks. Execution times on a t_k-type processor are 5, 5, and 6 time units for S_{i1}, S_{i2}, and S_{i3} respectively. The volume of data associated with the edge $d_{j1} = (S_{i1}, S_{i2})$ is 7 units and that with the edge $d_{j2} = (S_{i1}, S_{i3})$ is 6 units, and $D_{CR} = 1$. Now, let us say p_{k1} and p_{k2} are two t_k type-processors. Let S_{i1} be executed on p_{k1}. Now, since S_{i2} and S_{i3} can be executed in parallel,

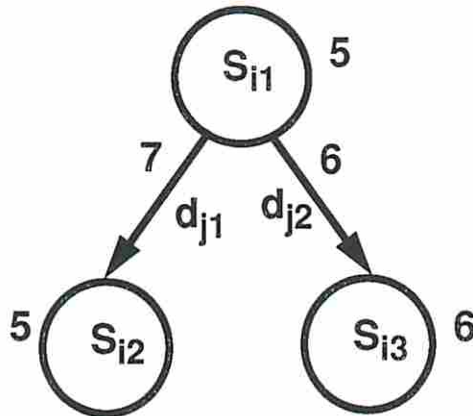


Figure 8.3: An Example to Illustrate Another Problem with Critical Path Estimation

S_{i2} can be executed on p_{k1} and S_{i3} on p_{k2} or vice-versa. In either case, the critical path delay is 17 time units. However, since all the three subtasks are being executed on the same type of processor, it is possible to execute them serially on the same processor, say p_{k1}, and the total delay in that case would be $5 + 5 + 6 = 16$ time units (no interprocessor communication delays). By serialization, we have been able to eliminate the interprocessor communication completely, and thus reduce the overall delay. It is clear that a simplistic use of critical paths is not directly applicable to finding a lower bound on the delay because the minimum possible delay of 16 time units does not correspond to a single path in the task graph. However, serialization would not always lead to the minimum possible overall delay. Let the volume of data associated with the edge d_{j1} be reduced to 2 units. Now, a parallel design, where S_{i1} and S_{i3} are mapped to p_{k1} and S_{i2} is mapped to p_{k2}, leads to an overall delay of $5 + 2 + 5 = 12$ time units. This example primarily demonstrates the tradeoff between parallelism and interprocessor communication. If exploiting parallelism leads to heavy interprocessor communication, serialization may be more appropriate. The following theorem illustrates this tradeoff more concretely. While this theorem

is useful only for specific task graphs, it leads to a more general theorem which will be stated subsequently.

Theorem 8.3.1.2 *Let a set of subtasks $T = \{S_b, S_e, S_l, S_r\}$ be given. Let the set of edges be given by $D = D_L \bigcup D_R$, where $D_L = \{d_{l0}, d_{l1}\}$ and $D_R = \{d_{r0}, d_{r1}\}$. The edges in the set D_L are specified as: $d_{l0} = (S_b, S_l)$; $d_{l1} = (S_l, S_e)$. Similarly, the edges in the set D_R are specified as: $d_{r0} = (S_b, S_r)$; $d_{r1} = (S_r, S_e)$. All the subtasks $S_a \in T$ are to be executed on the same type of processor; and for a given subtask $S_a \in T$, the corresponding execution time is t_a. The interprocessor communication delay associated with an edge $d_i \in D$ is given to be τ_i. A uniprocessor system can achieve the best possible completion time if and only if the following conditions are satisfied:*
(i) $\tau_{r0} + \tau_{r1} \geq t_l$
(ii) $\tau_{l0} + \tau_{l1} \geq t_r$
(iii) $\tau_{r0} \geq t_l$ *OR* $\tau_{l1} \geq t_r$
(iv) $\tau_{l0} \geq t_r$ *OR* $\tau_{r1} \geq t_l$

Proof: The given subtasks and edges can be represented by Figure 8.4. It is obvious from the figure that at most two processors should be used to perform the task. Let us say p_1 and p_2 are two processors (of the same type) capable of executing the subtasks.

The task completion time on a uniprocessor system (say, p_1) is given by:

$$TC_1 = t_b + t_e + t_l + t_r$$

Now, let us consider the execution on a two-processor system. Any such execution will involve exactly two data transfers. Without any loss of generality, let us assume that the subtask S_b gets on the processor p_1. Now, there are four possible cases:
Case I: S_e and S_l get executed on p_1, and S_r on p_2;
Case II: S_e and S_r get executed on p_1, and S_l on p_2;

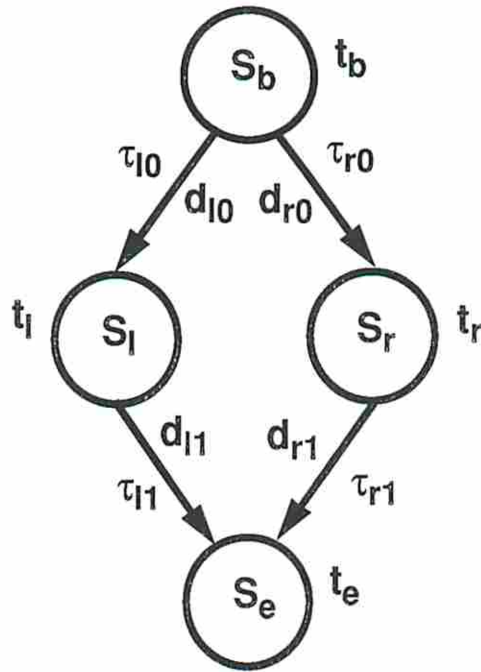


Figure 8.4: The Graph for Theorem 8.3.1.2

Case III: S_e and S_r get executed on p_2, and S_l on p_1;

Case IV: S_e and S_l get executed on p_2, and S_r on p_1.

Consider Case I. The task completion time is given by

$$TC_{2,I} = t_b + t_e + \max(t_l, \tau_{r0} + t_r + \tau_{r1})$$

The Case I execution scenario will be useful if $TC_{2,I} < TC_1$, or

$$\max(t_l, \tau_{r0} + t_r + \tau_{r1}) < t_l + t_r$$

If $t_l \geq (\tau_{r0} + t_r + \tau_{r1})$, then the above inequality is automatically satisfied. However, if $t_l < (\tau_{r0} + t_r + \tau_{r1})$, then the condition for the above inequality to be satisfied is

$$t_l > \tau_{r0} + \tau_{r1}$$

So, the Case I scenario can be used to improve the task completion time if $t_l > (\tau_{r0} + \tau_{r1})$. Similarly, it can be shown that the Case II scenario can be used to improve the task completion time if $t_r > (\tau_{l0} + \tau_{l1})$.

Now, consider Case III. The task completion time is given by

$$TC_{2,III} = t_b + t_e + \max(t_l + \tau_{l1}, \tau_{r0} + t_r)$$

This scenario can be used if $TC_{2,III} < TC_1$, or

$$\max(t_l + \tau_{l1}, \tau_{r0} + t_r) < t_l + t_r$$

If $(t_l + \tau_{l1}) \geq (\tau_{r0} + t_r)$, then the condition for the above inequality to be satisfied is $t_r > \tau_{l1}$. However, if $(t_l + \tau_{l1}) \leq (\tau_{r0} + t_r)$, then the condition is $t_l > \tau_{r0}$. So, the Case III scenario can be used if one of the following two conditions is satisfied:

$$\begin{aligned} \tau_{l1} &< t_r \leq \tau_{l1} + (t_l - \tau_{r0}), \quad or \\ \tau_{r0} &< t_l \leq \tau_{r0} + (t_r - \tau_{l1}) \end{aligned}$$

It is easy to see that one of the above is satisfied if and only if the following two constraints are satisfied:

$$\begin{aligned} \tau_{l1} &< t_r, \quad and \\ \tau_{r0} &< t_l \end{aligned}$$

So, the Case III scenario can be used to improve the task completion time if $\tau_{l1} < t_r$ and $\tau_{r0} < t_l$. Similarly, it can be shown that Case IV scenario can be used to improve the task completion time if $\tau_{l0} < t_r$ and $\tau_{r1} < t_l$.

Thus, a two-processor system can perform better if and only if at least one of the following conditions is satisfied:

1. $t_l > (\tau_{r0} + \tau_{r1})$ (Case I)

2. $t_r > (\tau_{l0} + \tau_{l1})$ (Case II)

3. $\tau_{l1} < t_r \; AND \; \tau_{r0} < t_l$ (Case III)

4. $\tau_{l0} < t_r \; AND \; \tau_{r1} < t_l$ (Case IV)

In other words, a uniprocessor system can achieve the best possible completion time if and only if none of the above conditions is satisfied. Q.E.D.

The above theorem can be generalized as follows.

Theorem 8.3.1.3 *Let a set of subtasks* $T = \{S_b, S_e\} \bigcup S_L \bigcup S_R$ *be given, where* $S_L = \{S_{l0}, S_{l1}, ..., S_{l(m-1)}\}$ *and* $S_R = \{S_{r0}, S_{r1}, ..., S_{r(n-1)}\}$. *Let the set of edges be given by* $D = D_L \bigcup D_R$, *where* $D_L = \{d_{l0}, d_{l1}, ..., d_{lm}\}$ *and* $D_R = \{d_{r0}, d_{r1}, ..., d_{rn}\}$. *The edges in the set* D_L *are specified as:* $d_{l0} = (S_b, S_{l0})$; $d_{lm} = (S_{l(m-1)}, S_e)$; $d_{li} = (S_{l(i-1)}, S_{li}), i = 1, 2, ..., (m-1)$. *Similarly, the edges in the set* D_R *are specified as:* $d_{r0} = (S_b, S_{r0})$; $d_{rn} = (S_{r(n-1)}, S_e)$; $d_{ri} = (S_{r(i-1)}, S_{ri}), i = 1, 2, ..., (n-1)$. *All the subtasks* $S_a \in T$ *are to be executed on the same type of processor; and for a given subtask* $S_a \in T$, *the corresponding execution time is* t_a. *The interprocessor communication delay associated with an edge* $d_i \in D$ *is given to be* τ_i. *A uniprocessor system can achieve the best possible completion time if and only if the following conditions are satisfied:*

(i) $\tau_{ri} + \tau_{rj} \geq \sum_{a|S_{la} \in S_L} t_{la}$; *where* $\tau_{ri} = \min_{k|d_{rk} \in D_R} \tau_{rk}$, *and* $\tau_{rj} = \min_{k|d_{rk} \in D_R - \{d_{ri}\}} \tau_{rk}$. *(ii)* $\tau_{li} + \tau_{lj} \geq \sum_{a|S_{ra} \in S_R} t_{ra}$; *where* $\tau_{li} = \min_{k|d_{lk} \in D_L} \tau_{lk}$, *and* $\tau_{lj} = \min_{k|d_{lk} \in D_L - \{d_{li}\}} \tau_{lk}$. *(iii) For any* i *(such that* $d_{li} \in D_L$*) and* j *(such that* $d_{rj} \in D_R$*),*
$(\tau_{li} \geq \sum_{k=0}^{j-1} t_{rk} \; OR \; \tau_{rj} \geq \sum_{k=i}^{m-1} t_{lk}) \; AND \; (\tau_{li} \geq \sum_{k=j}^{n-1} t_{rk} \; OR \; \tau_{rj} \geq \sum_{k=0}^{i-1} t_{lk})$.

This theorem can be proved in a way similar to Theorem 8.3.1.2.

The above discussion related to the idea of critical path consistently indicates that the major complication is caused by the interprocessor communication factor. When a heavy interprocessor communication exists, more and more serialization and inclusion of fewer processors in the system seem appropriate. If the amounts of data to be transferred at the edges of the task graph are large to begin with, the estimation of critical path becomes difficult. However, the initial partitioning of a task into component subtasks is usually performed with this problem in mind, and one of the goals is to keep the amounts of data transferred low. As we said earlier, for such "well-partitioned" tasks, our estimation of critical path will likely provide useful bounds for guidance of branch-and-bound. Also, Al-Mouhamed [AM90] has provided a loose and approximate lower bound on the completion time of a task graph for homogeneous systems; it seems that this work can be extended for heterogeneous systems to provide a lower bound (critical delay) on the completion time, and in fact, in the process some other bounds can also be generated. We briefly review these two ideas next.

8.3.1.1 Critical Path Estimation for "Well-Partitioned Tasks"

Let us define the phrase "well-partitioned task" first. Consider the set of subtasks and the set of data transfer edges for a given task T. For any edge $d_i = (S_{i1}, S_{i2})$, consider the subtasks S_{i1} and S_{i2}. Let the fastest processor type for S_{i1} be p_{fi1} and the corresponding execution time t_{fi1}, and those for S_{i2} be p_{fi2} and t_{fi2} respectively. Also, let $p_{i1,i2}$ be the fastest common processor type for the subtasks S_{i1} and S_{i2}; i.e., cumulative execution time $(t_{ci1} + t_{ci2})$ for the two subtasks is minimum on $p_{i1,i2}$-type processor. Let the delay associated with the remote transfer corresponding to edge d_i be τ_i. A given task T is considered *well-partitioned*, iff for each edge d_i, either $p_{fi1} = p_{fi2}$ or $t_{fi1} + t_{fi2} + \tau_i < t_{ci1} + t_{ci2}$. In other words, T is well-partitioned, if and only if for each communicating pair of subtasks, either the fastest processor types are same for the two subtasks

or cumulative execution time for the pair when each subtask is executed on its fastest processor type (including the communication time) is less than the corresponding cumulative execution time when the subtasks are executed on the fastest common processor type (no communication time).

It is easy to see that the intuitive critical path estimation approach described earlier will provide an accurate estimate for well-partitioned tasks.

8.3.1.2 Extension of Al-Mouhamed's Work

Al-Mouhamed [AM90] has developed some theory for scheduling precedence graphs with communication costs onto homogeneous systems and he derives some bounds. His theory revolves around two concepts:

- The notion of *Earliest Starting Time* for a given subtask.
- The notion of *Largest Delay* for a given subtask which does not increase the earliest completion time for the overall task. This notion of largest delay lends itself to the notion of *Latest Completion Time* for a given subtask.

Earliest Starting Time (EST) of a subtask can be computed once the EST's of all its predecessors are known. In an attempt to compute EST for a subtask, Al-Mouhamed finds out which of the predecessors should be executed on the same processor as the subtask under consideration. Essentially, predecessors with high communication get executed on the same processor as the subtask. Since EST's for subtasks having no predecessors are zero, we can compute EST's for all the subtasks. Based on EST's, the earliest completion time of the overall task can be computed. Having computed the earliest completion time of the overall task, largest delay possible (without affecting the completion time of the task) is computed for each of the subtasks, which also defines the latest completion time (LCT) for the subtask. Now, one could derive a subtask schedule interval without affecting the overall performance. Now, for each time-interval the number of subtasks (possibly executing concurrently in that time-interval)

is known. Largest number of subtasks that could possibly execute in parallel gives a bound on the number of processors. Similarly, Al-Mouhamed also derives bounds on the number of links using these ideas.

This theory can be extended to heterogeneous systems to derive bounds on the number of processors, number of links, cost and performance of the system. For homogeneous systems, the notion of EST automatically defines the notion of Earliest Completion Time (ECT) for a given subtask, since the execution time of a given subtask is fixed regardless of the processor on which it gets executed. However, for a heterogeneous system, since the execution time of a subtask depends on the type of the processor on which it gets executed, we need to explicitly define the notion of ECT. One crucial point to be kept in mind while developing theory for heterogeneous systems is that if the interprocessor communication is to be avoided between two subtasks then they have to be executed on the same processor which in turn implies that they have to be executed on the same processor type.

8.3.2 Estimation of the Upper Bound on the Task Completion Time

A tight upper bound on the task completion time will be given by the completion time achieved by the least expensive design. So, in the process of estimating an upper bound on the task completion time, we also estimate a lower bound on the cost of the system. Since the least expensive design will never consist of more than one of a type of processor, the following strategy can be used to estimate an upper bound on the task completion time:

1. Find the least expensive processor type t_{ch} which is capable of performing all the subtasks in the set T. The task completion time on a uniprocessor system consisting of a processor of the type t_{ch} is given by the sum of the

individual execution times of the subtasks on the processor (no interprocessor communication delays); i.e.,

$$TC_{uni} = \sum_{a|S_a \in T} D_{tS}(t_{ch}, S_a) \tag{8.4}$$

2. Now, consider only the processor types which are less expensive than the t_{ch} type and denote the set of such processor types by P_{ch}. Find all possible subsets of P_{ch} such that the sum of the costs of the processor types in the subset is less than C_{ch} (cost of a processor of type t_{ch}), and denote the set of such subsets by G_{ch}.

3. Now, for each subset P_{sch} in G_{ch}, estimate an upper bound on the task completion time and a lower bound on the cost of the system when the system consists of one processor each of the types in the subset. The upper bound on task completion time is estimated by assigning the largest possible execution times to the individual subtasks; i.e., for a particular subtask S_a, the execution time is considered to be $\max_{i|t_i \in P_{t,a} \cap P_{sch}} D_{tS}(t_i, S_a)$. Since the system consists of more than one processor, interprocessor communication delays have to be accounted for. Find the edges that may contribute interprocessor communication delays. For a given edge $d_i = (S_{i1}, S_{i2})$, if there is a *possibility* that the subtasks S_{i1} and S_{i2} may get executed on different processor types in the subset P_{sch}, then the interprocessor communication delay due to the edge d_i is added to the task completion time.

   Now, to estimate a lower bound on the cost of the system, a lower bound on the number of communication links needs to be estimated. For a given edge $d_i = (S_{i1}, S_{i2})$, if there are *no* processor types in P_{sch} which can perform both S_{i1} and S_{i2}, then the edge must lead to interprocessor communication. For each such edge, enumerate the set of communication links on which the communication may take place. Having found these sets of communication links, find the minimum set of communication links that must be created

to cover all the edges. The cardinality of the minimum set gives a lower bound on the number of communication links that must be present in the system. So, a lower bound on the cost of the system is given by the sum of the costs of the processor types in the set P_{sch} and the cost of creating the minimum number of communication links.

4. Now, we have estimated an upper bound on the task completion time and a lower bound on the cost of the system for each subset P_{sch} in G_{ch}. For a given P_{sch}, if the lower bound on the cost of the system is greater than C_{ch} (cost of the least expensive uniprocessor system), then discard the system consisting of processor types in P_{sch}. From among the rest of the P_{sch} subsets, find the one which has the largest upper bound on the task completion time. The larger of the largest upper bound and the uniprocessor task completion time (TC_{uni} defined above) gives an overall upper bound on the task completion time. Similarly, from among the P_{sch} subsets, find the one which has the smallest lower bound on the cost of the system. The smaller of the smallest lower bound and the uniprocessor system cost C_{ch} gives an overall lower bound on the cost of the system.

8.3.3 Estimation of the Upper Bound on the System Cost

A tight upper bound on the cost of the system will be given by the cost of the system which achieves the minimum task completion time. The minimum task completion time is given by the critical path delay. As we saw above, an accurate estimate of the critical path delay is not easy to obtain. Similarly, estimating a tight upper bound on the system cost is not easy. However, the bound reported in Section 8.2 can be tightened somewhat as follows.

Instead of including the costs of the processors having largest costs for each of the subtasks in the upper bound for the system cost, we should first find out

what processor types have a chance of being used to execute a given subtask in a system synthesized to achieve minimum task completion time. Since the goal is to minimize the completion time, attempt would be made to keep the individual subtask execution times as low as possible and eliminate the interprocessor communication delay wherever possible. With this in mind, it is easy to see that the processor type to be used for a given subtask should either be a processor type which minimizes the individual execution time for the subtask or a processor type which can be used for at least one of the predecessors or successors of the subtask in the task graph (thus eliminating some interprocessor communication delay). So, for each of the subtasks, we find a set of processor types where each of the types is capable of executing the given subtask as well as at least one of its predecessors or successors, and also include in the set the processor type which gives the minimum execution time for the given subtask. Now, for each of these sets, we find the largest-cost processor type in the set. The sum of these largest costs should be included in the upper bound for the system cost.

As we saw in this section, it is difficult to estimate tight bounds for the various design parameters, though it should be possible to compute reasonable bounds that can be used to guide the branch-and-bound. In Section 8.4, we look at some special case bounds.

8.4 Bounds for Special Cases

As we have seen, one of the reasons the estimation of tight bounds is difficult is the lack of knowledge about which subtask is going to be executed on which processor type. To simplify the problem, we assume that such information is available, and make an attempt to estimate bounds. The techniques described here, in fact, produce either quite loose bounds or have high computational complexity. They contribute to the understanding of the problem but are not all necessarily practical.

A useful and practical design problem is the synthesis of a system meeting a specified constraint on the task completion time. We now estimate a lower bound on the system cost given a constraint on the task completion time. Let the constraint value be T_{cons}. Since we know the processor type for each of the subtasks, we estimate a lower bound on the number of processors first. For each processor type, we find the subtasks to be executed on that type and the corresponding execution times. Now, if the overall task is to be completed within the time T_{cons}, the individual subtasks that are to be executed on a given processor type t_i must also be completed within the time T_{cons}. Now, the problem of finding a lower bound on the number of processors of a type t_i can be modeled as a *bin packing problem* [GJ79]. The execution times of the subtasks to be executed on a processor of type t_i are the "items" to be placed in a "bin," where the size of an item is the corresponding execution time value and the size of a bin is T_{cons}; and the goal is to pack all the items in as few bins as possible. The number of bins required gives a lower bound on the number of processors of the type under consideration if the task is to be completed within the time T_{cons}. As the bin packing problem is NP-hard [GJ79], it would be hard to compute this lower bound (unless the number of subtasks to be executed on a given processor type is small). Of course, a looser lower bound on the number of processors of the type is given by:

$$\left\lceil \frac{Sum\ of\ Individual\ Execution\ Times\ for\ the\ type}{T_{cons}} \right\rceil$$

Thus, we can get a lower bound on the number of processors for each type. Now, to estimate a lower bound on the number of communication links, we need to assign a type to each edge in the graph. The type assigned to an edge $d_i = (S_{i1}, S_{i2})$ is dictated by what processor types have been assigned to the subtasks S_{i1} and S_{i2} respectively. If the processor types assigned to S_{i1} and S_{i2} are t_{i1} and t_{i2} respectively, then the type assigned to the edge d_i would be (t_{i1}, t_{i2}). Now, group the edges based on the types assigned to them. For a group of edges

with the type (t_{i1}, t_{i2}), if $t_{i1} \neq t_{i2}$, we know that the data transfers related to the edges in the group must take place over communication links that go from processors of type t_{i1} to those of type t_{i2} (or let us say, over communication links of type (t_{i1}, t_{i2})). Now, we can estimate a lower bound on the number of communication links of type (t_{i1}, t_{i2}) in a manner similar to how we estimated a lower bound on the number of processors of a given type; i.e, by trying to pack the individual interprocessor communication delays into a bin of size T_{cons}. Thus, we can estimate a lower bound on the number of communication links that go from one processor type to another.

Now, we need to estimate a lower bound on the number of communication links that go between processors of the same type. Let us consider a set of edges E_{i1} with the type (t_{i1}, t_{i2}), where $t_{i1} = t_{i2}$. Find all possible subsets to the set E_{i1} such that all the edges in the subset form a connected component in the original task graph. For each such subset, find a lower bound on the number of processors of type t_{i1} required to complete the subtasks in the corresponding connected component within the time T_{cons} (since all the subtasks in the connected component are being executed on the processor type t_{i1}). Consider the largest of these lower bounds, say $n_{lb}(t_{i1})$. So, one of the connected components in the original task graph is going to be mapped to a set of at least $n_{lb}(t_{i1})$ processors of type t_{i1} in the synthesized system. Since a connected component in the original task graph must be mapped to a connected component in the synthesized system, there must be a connected component consisting of at least $n_{lb}(t_{i1})$ processors of type t_{i1} in the synthesized system. In other words, there must be at least $(n_{lb}(t_{i1}) - 1)$ communication links between processors of type t_{i1}. Thus, we can estimate a lower bound on the number of communication links that go between processors of the same type for each of the processor types. Since we have estimated a lower bound on the number of processors as well as the number of communication links, we have estimated a lower bound on the cost of the system.

Now, the question is whether the lower bound thus estimated is indeed the overall lower bound on the system cost, and the answer is, of course, no. The lower bound estimated is indeed the lower bound only when a given subtask is executed on a given processor type. So, to estimate an overall lower bound, one could estimate lower bounds for all possible combinations of subtasks and processor types. The lowest of all the lower bounds would give the overall lower bound. Obviously, such an approach may involve significant amount of computation just to estimate a lower bound on the system cost. One way to reduce the number of combinations to be tried would be to define a set of types of subtasks S_t. Then each subtask in the task graph would be assigned a type from the set S_t (based on the functionality requirements of the subtask). Now, we make the assumption: all the subtasks of the same type in the task graph would be executed on the same type of processor. As one would see, this assumption will reduce the number of subtask-processor type combinations to be explored. At the same time, one should also realize that the strategy based on this assumption does not necessarily produce the overall lower bound, and we may end up discarding some non-inferior designs in the process. Let us illustrate the problem with this assumption by an example.

Consider a task consisting of three subtasks S_1, S_2, and S_3 as shown in Figure 8.5. Let us say that they all are of the same type; i.e, they all have the same functionality requirements, and each one of them can be executed on two types of processors, p_1 and p_2, with execution times of 1 and 2 time units respectively. Let the cost of p_1-type processor be 20 units and that of p_2-type be 10 units. Let us say the task is to be completed within 2 time units. Now, if we follow the above assumption of using the same type of processor for all the subtasks of same type, we will end up using two processors of type p_1, with S_1 and S_2 executing on the first processor and S_3 on the second, and of course, the cost of the system will be 40 units. However, it is very easy to see that the constraint of 2 time units can be met by a system of cost 30 units, the system consisting of

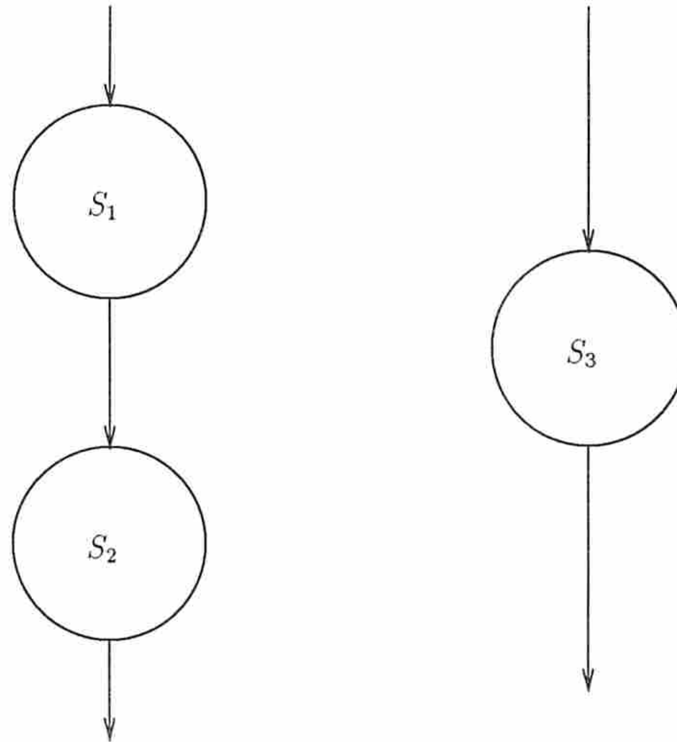


Figure 8.5: A Graph to Demonstrate Yet Another Problem with Bounds

one p_1-type and one p_2-type processors, with S_1 and S_2 executing on the p_1-type processor and S_3 on the p_2-type.

An assumption similar to the one discussed here is commonly used in datapath synthesis work [JMP88, JPP87, PPM86, PP86]. Most of the datapath synthesis efforts make an assumption that all the operations (nodes) of the same type in the data flow graph will be implemented by the same type of operator. For example, all the instances of the *addition* operation in the data flow graph will either be implemented by *ripple-carry adders* or by *carry-lookahead adders*, but not by a mixture of the two. It should be obvious now that even in the datapath synthesis domain such an assumption may prevent generation of the best possible design. The following theorem states this observation concretely.

Theorem 8.4.4 *Data path synthesis based on the assumption that all the instances of a type of operation in the data flow graph will be implemented by the*

same type of hardware operator does not necessarily generate the best possible design.

The proof can be easily performed using a hypothetical example (like the proof of Theorem 8.3.1.1). The hypothetical example will have a flavor of the example discussed above based on Figure 8.5.

The discussion on bounds has made it clear that the computation of tight bounds is a hard problem, and a significant amount of research needs to be performed in this direction. Though reasonable bounds can be easily computed that can be used to guide the branch-and-bound.

Chapter 9

Algorithmic/Heuristic Procedures for System-Level Synthesis

In this chapter, we explore some heuristic ideas and approaches for the system-level synthesis problem under consideration.

Since our synthesis approach is scheduling-driven, scheduling becomes one of the prime issues in deciding the quality of the approach. In high-level synthesis or data path synthesis also, scheduling is one of the prime issues. Several algorithmic/heuristic procedures have appeared in the literature for data path synthesis. In most of these efforts, scheduling is one of the major steps. In fact, it has been noticed that the contribution of the scheduling step to the quality of the design synthesized is enormous. For this reason, usually scheduling is the first step in data path synthesis. Taking inspiration from this, we realize that it is important to devise efficient scheduling heuristics for system-level synthesis. With this in mind, we first review some scheduling techniques at the system-level (or processor-level). Next we consider how to extend these heuristics to heterogeneous systems. Finally, we touch upon a new topic "synthesis of pipelined systems" at the end of this chapter.

9.1 Review of Scheduling Heuristics

At processor-level, scheduling problem has been researched quite extensively [Hu61, Gra69, FB73, CD73, ACD74, GGJ78, KN84]. Several solutions have been suggested. Most of the solutions concentrate on homogeneous systems. Earlier solutions did not consider communication overhead. The solutions revolve around the idea of *List Scheduling* (LS) heuristic. The most commonly used priority list in the list scheduling based solutions is organized in the decreasing order of levels of the subtasks, where level of a subtask is the length of the longest path from the subtask to a final subtask (i.e., a subtask having no successor). This heuristic is guaranteed to produce a schedule whose length t_{LS} is related to the optimal schedule length t_{opt} by the following expression:

$$t_{LS} \leq (2 - 1/n)t_{opt} \tag{9.1}$$

where n is the number of processors.

This upper bound on t_{LS} is called Graham's bound [Gra69, CD73]. In practice, the heuristic is found to produce near-optimal schedules. The scheduling problem is further complicated when communication overhead is significant. In case of no communication overhead, scheduling is performed to exploit maximum parallelism. However, in case of significant communication overhead, it may be appropriate to serialize certain subtasks in order to map them to the same processor so that interprocessor communication can be avoided. If the communication overhead is indeed very dominant, it may be worthwhile to *cluster* heavily-communicating subtasks into a group as a first step such that all the subtasks in a group get executed on the same processor. Detailed scheduling can be performed after this clustering step. Some clustering algorithms are reported in literature [GVY90, Sar89]. Some scheduling heuristics considering communication overhead are also reported:

- *ELS Heuristic [HCAL89]:* Apply LS heuristic ignoring communication overhead (denote the length of schedule so obtained by t_{LS}), and then add necessary communication delays to the schedule obtained (denote the length of the extended schedule by t_{ELS}). The difference between t_{ELS} and t_{LS} is upper bounded by the total maximum communication requirement (i.e., when all the edges lead to interprocessor communication delay).

- *ETF Heuristic [HCAL89]:* ETF (Earliest Task First) is an event-driven heuristic - the earliest schedulable subtask is scheduled first. At each step, the algorithm schedules a subtask T on a processor P if the earliest starting time of T on P is the smallest among all the schedulable subtasks and all the available processors. The length of an ETF schedule, t_{ETF}, is bounded by the sum of Graham's bound for list scheduling and the communication requirement C along one chain of subtasks that can be calculated using an algorithm [HCAL89]:

$$t_{ETF} \leq (2 - 1/n)t_{opt} + C \tag{9.2}$$

  where t_{opt} is the optimal schedule length obtained by ignoring the communication overhead and n is the number of processors.

- *MH Heuristic [ERL90]:* In this heuristic, leveling is performed assuming all the edges will lead to interprocessor communication delay. At each step, the subtask having highest level is assigned to a processor on which it finishes earliest. No bounds have been proven on the schedule length produced by this heuristic. MH heuristic has been applied to heterogeneous systems also, where different processor types have different speeds; so execution times of various subtasks uniformly increase or decrease when they move from one processor type to another. This is the reason a subtask is assigned to a processor on which it finishes earliest (thus taking into account the execution time of the subtask on the processor under consideration).

However, the issue is how to perform leveling when execution times vary arbitrarily from one processor type to another.

These heuristics seem to perform reasonably well in practice. We would like to investigate the effectiveness of these scheduling heuristics for the synthesis problem under consideration. Since our synthesis problem is geared towards truly heterogeneous systems where execution times can vary arbitrarily from one processor type to another, the first problem we face is how to extend these heuristics to heterogeneous systems (as the heuristics essentially deal with homogeneous systems).

9.2 Extending Heuristics to Heterogeneous Systems

Essentially, list scheduling based heuristics have two primary components:

- *Subtask selection:* how to select the next subtask to be scheduled
- *Processor selection:* how to select the processor to execute the selected subtask

In level-based list scheduling, the priority of a subtask is given by its level. The level of a subtask defines the amount of time that must elapse after the start of the subtask before the overall task is completed. In case of a homogeneous system with negligible communication overhead, it is fairly straightforward to compute the amount of time that must elapse to complete the overall task after the start of a given subtask. But how does one compute this value in case of a heterogeneous system with significant communication overhead? To answer this question, let us look at how this value is computed for scheduling heuristics for homogeneous systems with significant communication overhead. ELS heuristic completely ignores communication overhead in subtask selection (uses LS as a

first step). The rationale behind such a strategy would be that there is a possibility that edges do not lead to interprocessor communication (i.e., optimism), and so we call it *optimistic leveling.* ETF heuristic does not perform subtask selection and processor selection as separate steps. A subtask-processor pair is selected such that some subtask gets scheduled as soon as possible. So, there is no explicit leveling involved in this heuristic. MH heuristic performs *pessimistic leveling*, as it is based on the pessimistic rationale that all the edges may indeed lead to interprocessor communication. So, we see that the leveling problem becomes hard in presence of communication overhead due to the uncertainty that a particular edge may or may not lead to interprocessor communication. It goes without saying that the leveling problem is further complicated in heterogeneous systems due to introduction of further uncertainty about the delay associated with a node (subtask) which becomes dependent on the type of the processor to which the node would be mapped. In any case, let us see how we could perform leveling for heterogeneous systems:

- *Optimistic Leveling:* Here, we hope for the best possible scenario; i.e., each subtask is executed on its fastest processor type and none of the edges lead to interprocessor communication. So, leveling is performed with each node assigned the smallest possible execution time and each edge assigned a value of zero.

- *Pessimistic Leveling:* Here, the worst possible scenario is considered. So, leveling is performed with each node assigned the largest possible execution time and each edge assigned an interprocessor communication delay based on the amount of data on the edge.

Having looked at optimistic and pessimistic leveling, one would wonder why not go for somewhat *realistic leveling.* Along with the above two extreme leveling strategies, we think it would be useful to have one realistic leveling strategy. The idea behind leveling is to come up with the priority list for list scheduling. One intuitive way to order such a priority list would be to arrange the subtasks

in increasing order of LST (Latest Starting Time), where LST of a subtask is defined as the latest time by which the subtask must start executing if the overall task has to be completed at the earliest possible time (LST defines a *deadline* if optimal schedule is desired, and subtask having earlier deadline should be given a higher priority). As we mentioned in Section 8.3.1.2, Al-Mouhamed [AM90] has developed some theory and we discussed how to extend it to heterogeneous systems. Such a theory would involve definition of parameters like EST, ECT, and LCT for each of the subtasks. We can also define a parameter called LST using similar ideas. Then we can use this parameter in building the priority list for list scheduling. We call this scheduling strategy "realistic leveling strategy."

Having looked at subtask selection strategies in list scheduling, let us look at processor selection strategies. Generally, in list scheduling, one selects a processor on which the subtask can be started earliest. This makes sense in case of homogeneous systems since "starting earliest" implies "finishing earliest." Since in heterogeneous systems, execution time of the subtask may depend on the processor selected, it seems more reasonable to select a processor on which the subtask can be finished earliest (as done in MH heuristic).

In summary, following are some useful scheduling heuristics for heterogeneous systems with interprocessor communication:

- *Heuristic I:* List scheduling based on optimistic leveling for subtask selection and "finishing earliest" strategy for processor selection.
- *Heuristic II:* Similar to Heuristic I with "optimistic leveling" replaced by "pessimistic leveling."
- *Heuristic III:* Similar to Heuristic I with "optimistic leveling" replaced by "realistic leveling."
- *Heuristic IV:* This heuristic is based on an extension of ETF heuristic for homogeneous systems. At each step, the algorithm schedules a subtask T

on a processor P if the earliest finishing time of T on P is the smallest among all the schedulable subtasks and all the available processors.

Looking at Heuristic III, one thought comes to mind naturally. Realistic leveling involves computation of parameters like EST, ECT, LCT, and LST for each of the subtasks. Computation of these parameters involves determination of which of the predecessors of a given subtask should be *merged* together with the subtask (i.e., executed on the same processor as the subtask). One could argue "if the information regarding which subtasks should be merged together is already being derived, then why not directly use this information to perform scheduling?" There are two problems in trying to do so. First of all, the estimation of these parameters is done under the assumption that there are as many processors available as desired, which means the technique can not be directly used if the number of processors is restricted. Secondly, estimation of EST may dictate that a given subtask should be merged with two of its successors; then the issue is which one it should be merged with. Obviously, this problem makes it hard to use this technique for scheduling. Of course, one solution to this problem is to indeed merge the subtask with both the successors, in which case the subtask is actually executed twice - once on each of the processors executing the two successors respectively, and thus completely avoid the interprocessor communication. Such an approach would pay off if there are large amounts of data transferred to both the successors. The notion of allowing a subtask to be executed more than once in order to avoid heavy interprocessor communication is referred to as *recomputation* [KL88]. We have not talked about recomputation so far in this thesis. In a given situation, recomputation may or may not be allowed. If recomputation is indeed allowed, then the technique could be used to perform scheduling under the assumption of availability of unlimited number of processors. Essentially, the scheduling is done as follows. Each subtask is scheduled as early as possible (obviously, it can't be earlier than its EST parameter). So, it is executed as soon as all its predecessors have been executed and necessary data has been

received. While estimating EST parameter, it would have been found that certain predecessors should be merged with the subtask; all such mergings should be performed while scheduling. Let us call this technique *EST-based heuristic.* It is our conjecture that schedule length obtained by EST-based heuristic is within twice the optimal schedule length for homogeneous systems. It will be worthwhile to consider this heuristic for homogeneous as well as heterogeneous systems.

In this section, we have developed several heuristics that can be used for scheduling of heterogeneous systems. Finally, the issue is how to use these scheduling heuristics for synthesis. Usually, scheduling is done given a certain number of processors etc. In synthesis, we may or may not have such information a priori. In data path synthesis, scheduling has been used successfully; similar ideas can be used for the synthesis problem under consideration.

9.3 Synthesis of Pipelined Designs

So far, we have only talked about synthesis of non-pipelined designs in this thesis. Eventually, one would like to consider pipelining at system-level. Park [Par85] has developed a theoretical basis for synthesis of pipelines at RT-level. It would be worthwhile to investigate if such a theory can be extended/developed for processor-level designs we are considering. Park did not consider communication cost or delay; so it is possible that his model does not hold when such effects are considered.

Before synthesizing pipelined designs or developing theory for the same, one must understand what exactly such a design or system is. One must define what one means by a pipelined system consisting of processors and communication links, for example. We give such a definition here for a pipelined system consisting of processors and communication links.

Definition 9.3.1 *A processor-link pipelined system consists of p stages, (S_1, S_2, ...,S_p). A stage S_i consists of a set of processors P_i and a set of communication*

links L_i. For each stage S_i, a set $I_i \subseteq (P_i \bigcup L_i)$ and a set $O_i \subseteq (P_i \bigcup L_i)$ are defined. An element $X \in I_i$ represents a piece of data input to X (and thus to stage S_i also) either from some previous stage S_j $(1 \leq j < i)$ or from external environment. Similarly, an element $Y \in O_i$ represents a piece of data output from Y (and thus from stage S_i also) either to some further stage S_j $(i < j \leq p)$ or to external environment. An element $X \in I_i$ represents a piece of data input from the previous stage S_j $(1 \leq j < i)$, if and only if there is an element $Y \in O_j$ representing a piece of data output to stage S_i; and $X \in P_i$ iff $Y \in L_j$. A parameter T_{in} specifies the time spent in each stage, and is called initiation interval (as a new set of data can enter the system every T_{in} time units).

The most crucial point in the above definition is the fact that when two stages exchange data, a communication link must be involved at the interface of the two stages. Now, given a task graph, the synthesis problem is to design a pipelined system well-suited for the task.

As a first step, let us see how one could go about synthesizing a pipelined system with optimal performance (i.e., minimum initiation interval). First we need to determine the minimum possible initiation interval. This can be done as follows:

1. Find the fastest execution times for each of the subtasks, and then the largest among these fastest execution times (call this largest value T_{fl}). Set minimum initiation interval (MII) to be T_{fl}.

2. Now, find all the interprocessor communication delays corresponding to the edges, and sort the edges in decreasing order of interprocessor communication delays. Initialize each node to be a separate cluster.

3. Delete edges with delay not greater than MII from the sorted list.

4. Select the largest delay edge from the list and try to eliminate it. Find out the source-destination pair for the edge. Try to merge the source and

destination clusters. If merging not possible (no common processor type available), then set MII to be the delay corresponding to this edge. Also, if cumulative execution time for the merged cluster larger than edge delay, then again set MII to be the delay corresponding to this edge. Otherwise, if cumulative execution time for the merged cluster larger than the current MII, then set MII to be the cumulative execution time. Go to step 3.

Once initiation interval is known, a simple way to synthesize a pipelined system could be as follows. Perform scheduling using one of the heuristics developed for the non-pipelined case. Schedule as many subtasks as possible in the current stage as long as the execution time for the stage does not exceed the desired initiation interval. After finishing one stage, we need to add necessary communication links either in the current stage or the next stage (if we can not fit the data transfer delay within the current stage without exceeding the initiation interval). After adding the necessary communication links and delays, continue scheduling for the next stage. Stop when the whole graph is scheduled.

The procedure outlined above is fairly simplistic in nature. Obviously, much research needs to be done to develop sophisticated procedures, and to develop theory which can help us understand this problem better. Such a pipeline synthesis problem is not researched at all so far. In fact, even for homogeneous systems, the problem is not researched. Questions such as "if we know the initiation interval, what can we say about the number of stages in the pipeline?" should be addressed. Finally, more complex pipelined structures should be considered. For example, each stage could consist of several sub-stages, and in that case resource sharing could occur between different stages; e.g., sub-stage 1 of stage 1 could share resources with sub-stage 2 of stage 2.

Chapter 10

Conclusion and Future Research

The focus of this thesis is on application-specific multiprocessor systems. As discussed in Chapter 1, application-specific multiprocessor systems can provide better performance and/or cost-effectiveness, as the application and the system can be very well-matched. More and more such systems are and will be needed. The thesis addresses the problem of design of application-specific multiprocessor systems. A synthesis approach has been advocated for the problem. SOS is a formal synthesis approach developed that can be used for design of application-specific multiprocessor systems.

The SOS approach is quite general and flexible. Given the application task characteristics and the system (to be designed) characteristics, the approach involves creation of a formal model using mathematical programming. The formal model captures the task characteristics and the system characteristics as well as the system design rules to be followed. The model encompasses all the relevant tradeoffs that must be considered to synthesize an optimal design. Indeed SOS is geared towards generating optimal or non-inferior designs. Solution of the mathematical programming model produces an optimal design. Static scheduling and allocation is performed in the SOS approach, in order to provide a better match between the application and the system and thus produce an overall optimal design. SOS synthesizes heterogeneous systems, as they can better match the application and hence tend to be more efficient as well as cost-effective.

The SOS approach has been validated by developing applicable mathematical programming models for different system models and performing synthesis experiments with them. The practicality of the approach has also been addressed, and some techniques for reducing the computer time to solve the SOS mathematical programming model have been investigated. The techniques show significant improvement is achievable through them. An attempt is also made to develop some theory and bounds with the dual goal of making SOS more practical as well as understanding the synthesis problem better. Some heuristic ideas have been outlined for the synthesis problem, and some scheduling heuristics have been developed for heterogeneous systems. Finally, the subject of "system-level pipelining" is introduced.

In this chapter, we present a critical evaluation of the research described in this thesis. We also propose areas for future research. The research described can be classified into four major areas:

- SOS modeling approach
- Runtime improvement for SOS models
- Theory and bounds development
- Development of heuristics

Below we discuss each area.

10.1 SOS Modeling Approach

10.1.1 Contributions

The prime contribution of the research described in this thesis is the SOS modeling approach, where the essential properties necessary to ensure correct system design for a given application task can be expressed as a system of algebraic relations. The modeling approach provides a unified means of expressing the

decisions and tradeoffs involved in designing a multiprocessor system for a given application. Various factors (e.g., IPC, load on a processor) affecting multiprocessor system design can be accounted for in the modeling approach. Complicated interrelationships exist between the system and the order in which various events occur in the task execution; the modeling approach is capable of expressing them in the form of algebraic relations. The modeling approach is also capable of handling arbitrary constraints on various aspects of the system imposed by the designer.

Use of algebraic relations and mathematical programming for the modeling offers a very powerful, general and flexible technique that can be applied to several varying design situations and design styles. Adequacy and generality of the modeling approach has been verified by developing several different mathematical programming models for different types of design situations. In order to solve the models, some linearization techniques have been developed and it has been shown that the models can be converted into MILP problems. Adequacy of the approach is further demonstrated by solving the MILP models using a branch-and-bound program and thus producing some correct and optimal system designs.

Of course, the strength of SOS is in dealing with parallelism among major subtasks of the application. The first SOS model developed is described in Chapter 3, where one of the primary characteristics of the systems is the use of point-to-point interconnection style between the processors. Point-to-point interconnection can be helpful in providing a better match between the application and the system, and hence such systems may have better performance. Several experiments have been performed with the point-to-point model and are reported in Chapter 4. The experiments clearly indicate that SOS modeling approach can be used to perform synthesis. Different systems have been synthesized for a given application, depending on the cost-performance requirements imposed by the designer. Some tradeoff studies have also been performed using the model

which lead to intuitive results. About the role of inter-subtask communication, it is found that heavy inter-subtask communication leads to designs with fewer processors and multiprocessing is more useful only when inter-subtask communication is reasonable. Such results show the validity of the modeling approach. Of course, it also implies that SOS can be very useful in evaluating tradeoffs and performing such studies. In Chapter 5, an SOS model for bus-style interconnection has been developed and used in synthesis experiments. Once again, synthesis of different systems depending on the cost-performance requirements imposed validates the approach.

In Chapter 6, the SOS approach has been used to model memory as an explicit design parameter in system design. First, the SOS model is extended to include costs associated with memory requirements during the computation of subtasks; i.e., memory required for code as well as temporary computation data of subtasks. Several synthesis experiments have been performed with the extended model, which verify the capability of the modeling approach to include memory as a design parameter. The results seem to imply that the systems with larger number of processors would tend to become inferior when memory costs are considered, and that systems with fewer number of processors would tend to provide better overall cost-performance ratio. Such results clearly make it imperative to consider memory costs explicitly during system design. Of course, we have demonstrated that the SOS modeling approach is capable of dealing with this challenge. Finally, we showed how to extend the SOS model to consider memory requirements due to buffers for interprocessor communication, and developed some linearization techniques for the extended model. Although the basic capability of the modeling approach to handle such complex issues is demonstrated, it is clear that the model development begins to look complicated as we go to more complex issues. Particularly, the linearization process becomes harder.

As we attempt to model more and more complex issues using the SOS approach, it is conceivable that eventually we may have to resort to very complex

linearization techniques. Of course, that is not the only direction to adopt. Linearization is required to solve the model as an MILP problem. The other possible direction is to use some nonlinear programming techniques to solve the SOS mathematical programming model. One way or the other, it seems that the modeling approach can be exploited to handle several complex issues not explicitly mentioned in the thesis.

All in all, the various models developed and their use in generating several correct and optimal system designs (as demonstrated by experimental results) show that the modeling approach can be used to develop correct, complete and consistent models to be used in different design situations.

In summary, the SOS modeling approach provides a systematic procedure that can be used in design of dedicated multiprocessor systems. Here are some distinguishing features of the procedure and the research presented in this thesis:

- automatic design of the multiprocessor system itself, not merely the mapping of tasks onto a fixed system,
- synthesis of heterogeneous systems, in terms of the functionality and the cost-speed characteristics of the processors, and
- synthesis of custom, heterogeneous interconnection schemes.

Apart from being useful in synthesis/design, another contribution of the SOS synthesis procedure will be in understanding the system-level design process itself. Particularly, the experiments performed with SOS will be helpful in this. Experimental results can indicate what is important. For example, one such result was the finding that it is important to consider memory costs explicitly. Another use of SOS is in handling partial system design specification and verification. We describe these aspects in the next section.

10.1.1.1 Use of SOS in Partial Design and Verification

The SOS approach is very versatile. Apart from complete system-level design, it has other uses as well. It offers significant degree of control to the designer in the design process. The designer does not have to use SOS for complete system design only. The designer may already have a partial system design, and the goal may be to complete the design. SOS can be used in such a scenario. The partial design can be specified to SOS, and it will generate the rest of the design. This is easy to accomplish in the SOS modeling approach as pre-existence of part of the design essentially implies some of the design decisions have already been made which in turn implies that some of the binary variables in the SOS model have been fixed to a specific value, and hence the model gets reduced to a simpler one. In fact, the simpler model generated would be easier to solve (in terms of runtime) also as it has fewer variables. Another control offered to the designer is in terms of arbitrary constraints that can be specified. For example, if one of the outputs of a particular subtask must be available by a certain time (as it may be required by some other system which is working in cooperation with this system), then such a constraint can be specified in SOS at design time. The ability of SOS to allow designer control (in terms of partial design and arbitrary constraint specification) makes it well-suited for making engineering changes in existing systems as well as for iterative design of complete systems.

Finally, SOS can be used for verification. The problem here is that the system is already designed, and the designer wishes to verify if the system can be used to achieve a certain level of performance for a given application task. Again, SOS modeling can be used. Since the system is already present, several binary variables would disappear from the model and a performance requirement would be specified as a constraint in the model. Now, the goal would be to see if a feasible solution to the generated model exists. If it does, we have verified that the system can provide the desired performance; otherwise it can not. Again, the model generated would be much easier to solve as it has fewer variables in the

first place, and secondly we are looking for a feasible solution (not an optimal one) in this case.

The capability of SOS to allow its use for partial design and verification situations makes it applicable to even very large applications and systems where full-fledged system design using SOS may not be possible due to large MILP models generated. However, partial design and verification problems will have smaller models to solve. Thus, the complete system can be designed using some sort of iterative approach based on SOS. Of course, the overall system may not be optimal in this case, but a reasonably good design could be found.

10.1.2 Future Research

10.1.2.1 Enhanced SOS Models

As we mentioned, the SOS modeling approach is quite general; and we demonstrated this generality by developing a few models. It must be emphasized that the approach could be applied to model several other design scenarios. One research direction would be to explore this generality further. Here we outline a few ideas in this direction.

The modeling approach could be applied to other interconnection styles. Although we believe that for application-specific systems, point-to-point interconnection is probably more useful as it can provide better performance, in general, one would like to have an ability to explore other interconnection styles. For example, we developed a model for bus-style interconnection in Chapter 5; however, we assumed that only one type of bus was available and was by default chosen. In general, n types of buses may be available, and the design involves a choice in the bus type. Model developed should include such a choice. Of course, in general, instead of having only one bus in the final system designed, one could have multiple buses and different processors could be connected by different buses. Again, the model should allow such possibilities. In fact, for point-to-point interconnection also, different types of links may be available, and

the model should reflect the choice. Another possible interconnection style is a star form. One would also like to consider the possible presence of a mixture of different interconnection styles in the system to be designed.

More complex memory issues/structures need to be addressed. For example, shared-memory systems could be modeled. Other hierarchical memory structures and cache have to be considered.

Models could be developed to handle some other issues as well. For example, the specific model of Section 3.2 assumes overlap between computation and I/O operations. A model could also be developed for the situation when such an overlap is not possible. Another issue deals with data format. Different processors may have different data formats. In such a case, when they communicate with each other, data format conversion is performed. Obviously, there would be some cost and delay associated with the process of conversion. Model should take care of these aspects. Another issue concerns "recomputation" discussed in Section 9.2. In the models developed in this thesis, each subtask is executed once and only once. If recomputation is allowed, then the models need to be developed with that in mind. Peripheral devices and other system resources need to be considered as well. If processors require some peripheral devices or other resources during the computation of subtasks, then the model should represent whether such resources should be shared or not, and if yes then how much sharing. Such decisions should be included in the model. Finally, issues of reliability of the system may need to be considered. To incorporate some degree of reliability in the system, homogeneity may be desirable. However, performance may dictate heterogeneity. This tradeoff could be reflected in the model.

We have outlined some issues that could be of relevance in a given design situation. Models such as these need to developed and experimented with. The list given above is by no means exhaustive. In general, there may be many more such issues. Of course, what is relevant is dictated by the given design problem, and the appropriate model is chosen based on that.

10.1.2.2 Other Research Directions

We consider some other research directions here. The starting point for SOS is a partitioned task graph. In order to build a complete synthesis system that can be used for design of application-specific multiprocessor systems, we need to develop effective techniques that can be used for partitioning a given application task into subtasks. In the USC project, CHOP [Kuc91] is geared to perform partitioning for custom hardware ASICs. One possible effort would be to enhance CHOP to handle programmable hardware elements (processors) as well and hence make it suitable to perform partitioning desirable for SOS. The other piece of information required by SOS is how effective a given processor is for a given subtask. Development of techniques for an accurate evaluation of such parameters is another research direction. Of course, one way to do this evaluation is through simulation.

10.2 Runtime Improvement for SOS Models

10.2.1 Contributions

Experimental results with the SOS models indicate that the approach is usable up to a certain size application and processor set. Beyond a point, we have to develop better solution strategies for the MILP model. We investigated some techniques for efficient branch-and-bound exploration for solving the MILP model.

First we looked into some "branching strategies." One of the branching strategies investigated was priority assignment to binary variables. The motivation behind priority assignment is to be able to use design knowledge and emulate an expert design strategy. We concluded that expert design strategy is not very well understood. Then we developed some simplified strategies for priority assignment based on intuition and motivation to strengthen the constraint system. The experiments with the simplified strategies indicate the more intuitive strategy of priority assignment in the (σ, α, ϕ, rest) ordering for the MILP model of

Section 3.2 to be promising in runtime reduction. The other branching strategy investigated was branching on a group of binary variables. The prime motivation behind this strategy is that some variables are very closely related to each other and it is important to be able to fix them simultaneously. Some such groups were identified in the MILP model of Section 3.2 again, and some experiments were performed with some of the group combinations. However, results of these experiments were inconclusive. Then priority assignment and group branching were jointly investigated. This investigation led to the conclusion that group combination 1 (σ-Group, γ, δ-Group) and (σ, α, ϕ, rest) ordering together show a great promise for runtime improvement. Some constraint reformulation was proposed to develop more useful branching groups.

Having investigated branching strategies, we looked into "constraint satisfaction techniques." We showed how Sidebottom's technique can be used to force values of binary variables quickly. Then we performed experiments with Bonsai (a program incorporating Sidebottom's technique) whose results were inconclusive. Bonsai was then investigated in conjunction with branching strategies. This investigation led to the conclusion that a constraint satisfaction technique, group combination 1, and (σ, α, ϕ, rest) ordering used together have maximum potential for runtime improvement. Further experimentation confirmed this belief.

In summary, we have developed some strategies to be used in conjunction with branch-and-bound that show tremendous potential for runtime improvement for MILP solution. The degree of runtime improvement is much more significant for the larger example. We hope that use of these strategies in solving the MILP model will make SOS much more practical for larger examples.

10.2.2 Future Research

Strategies developed for efficient branch-and-bound exploration need to be further investigated in depth. Attempt should be made to discover more such

strategies. For example, some unintuitive priority assignments should be investigated. In fact, it would be worthwhile to investigate different permutations for priority assignment. Attempt should be made to identify more branching groups and discover useful ones by experimentation. Different combinations of various groups identified should be tried. If necessary, some constraints should be reformulated to discover better branching groups. Other constraint satisfaction techniques should be investigated [Nad89]. Finally, different combinations (and permutations) of various strategies should be investigated.

The idea of dynamic priorities proposed in Section 7.3.1.5 should also be investigated. The goal should be to emulate the expert design strategy through such a dynamic approach. As we mentioned in Section 7.3.1.5, in order to effectively implement dynamic priorities, one requires an ability to look at the partial design constructed so far at a given branching step. With this in mind, it would be a useful effort to develop a program that can output the partial design at an intermediate step during the branch-and-bound exploration. With the help of such a program, one can look at the partial design at the time of branching and reassign priorities if necessary. Thorough investigation of static and dynamic priorities along with branching groups could provide sufficient design knowledge which could be used to eventually build an expert system which solves the MILP model by guiding the branch-and-bound procedure. This expert system should actually emulate an expert design strategy.

Another direction of research would be exploration of nonlinear programming techniques for solving SOS models. As we saw in Section 7.3.1.3, linearization introduces some weakness in the constraint system, and leads to serious problems in terms of solving the model. So, it would be worthwhile to see if nonlinear techniques for solution could be more effective.

Finally, if the designer is willing to give up optimality, some heuristic techniques could be developed that can be used in conjunction with branch-and-bound to solve the model.

10.3 Theory and Bounds Development

10.3.1 Contributions

Some theory has been developed to compute bounds on various design parameters. Such bounds can be useful in reducing the search space and thus improving the computational performance of branch-and-bound. Bounds on system cost, performance, number of processors, and number of communication links are developed. Associated difficulties in estimating tight bounds are identified. Critical path estimation is shown to be hard in presence of heavy inter-subtask communication; however, a well-partitioned task is defined for which such an estimation becomes possible. Although estimation of tight bounds is shown to be hard, reasonable bounds can be computed to be used with branch-and-bound.

10.3.2 Future Research

The theory and the bounds developed need to be further refined. Further investigation is needed to see if tighter bounds can be generated. Also, the bounds developed need to be incorporated into the branch-and-bound exploration to investigate if the computational performance is improved. The theory developed is for the simplified problem described in Section 8.2. So, it needs to be extended to more general problems as described next.

10.3.2.1 Consideration of Subtleties

The simplified problem of Section 8.2 assumes that a subtask requires all the inputs before it can start and all the outputs from the subtask become available only after the subtask is completed. In the original statement of the problem in Section 3.2, we had said that subtasks did not require all the inputs before starting their execution and they might produce some outputs even before their completion. How do we take this complication into account in the theory?

One way to accomplish this would be to split a given subtask into q subtasks where q is the sum of number of inputs and number of outputs of the subtask. As an example, let us say a subtask S_1 has two inputs $i_{1,1}$ and $i_{1,2}$, and two outputs $o_{1,1}$ and $o_{1,2}$. Let us further say that S_1 requires $i_{1,1}$ when 25% of the subtask is accomplished, requires $i_{1,2}$ when 50% is accomplished, produces $o_{1,1}$ when 75% is accomplished, and produces $o_{1,2}$ when 100% is accomplished. S_1 could be split into 4 mini-subtasks: S_{1a} (first 25%), S_{1b} (second 25%), S_{1c} (third 25%), S_{1d} (fourth 25%). Input $i_{1,1}$ goes to mini-subtask S_{1b}, and input $i_{1,2}$ goes to mini-subtask S_{1c}; output $o_{1,1}$ is produced by mini-subtask S_{1c}, and output $o_{1,2}$ is produced by mini-subtask S_{1d}. In addition, each of the mini-subtasks produces a dummy output which goes to the next mini-subtask in sequence (i.e., S_{1b} sends an output to S_{1c}). Now, each of the mini-subtasks requires all the inputs before it can start and all the outputs from a mini-subtask become available only after the subtask is completed. Now, it should be possible to apply some of the theory developed in Chapter 8. However, one has to keep in mind that all the mini-subtasks of a given subtask must be executed on the same processor, and they should all be executed immediately after one another. This constraint would require some modifications in the theory. This needs to be investigated.

10.3.2.2 Extension to More Complex Design Situations

Theory should be extended to more generalized design problems; e.g., to systems connected via a bus (instead of point-to-point). Also, development of theory should be investigated for systems where memory is also a crucial parameter.

10.4 Development of Heuristics

10.4.1 Contributions

Some scheduling heuristics have been developed. First, a critical review of the existing multiprocessor scheduling heuristics is provided which essentially deal

with homogeneous systems. Then it is shown how to extend the heuristics to heterogeneous systems. Different kinds of leveling strategies are proposed, and four different scheduling heuristics are developed. Also, one heuristics based on "recomputation" is proposed.

Finally, system-level pipelining is introduced. A simple procedure to determine minimum initiation interval is proposed; and then a simple procedure using scheduling heuristics is outlined for synthesis of a pipelined system with optimal performance.

10.4.2 Future Research

The scheduling heuristics developed need to be investigated in two directions. First of all, programs should be written which implement these heuristics, and large number of examples should be run through the programs to see how good the results are. The same examples could be solved optimally using the MILP approach, and then the heuristic and optimal results could be compared. Secondly, these heuristics should be studied from a theoretical point of view, and attempt should be made to prove bounds on how far the results would be from the optimal. It should be investigated if the theory developed in Chapter 8 could be combined with these scheduling heuristics to create useful algorithmic/heuristic procedures for system-level synthesis problem.

Finally, system-level pipelining being an important issue, development of theory and synthesis procedures for pipelined designs should be investigated. The problem should be investigated for homogeneous as well as heterogeneous systems. Since the problem has not been researched, significant amount of work needs to be done here.

Reference List

[ACD74] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, Dec. 1974.

[AJ88] R. Agrawal and H. V. Jagadish. Partitioning techniques for large-grained parallelism. *IEEE Transactions on Computers*, 37(12):1627–1634, Dec. 1988.

[AM90] M. A. Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Transactions on Software Engineering*, 16(12):1390–1401, Dec. 1990.

[Amd67] G. Amdahl. The Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[Bok79] S. H. Bokhari. Dual Processor Scheduling with Dynamic Reassignment. *IEEE Transactions on Software Engineering*, SE-5(4):341–349, July 1979.

[Bok81] S. H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, SE-7(6):583–589, Nov. 1981.

[Bok87] S. H. Bokhari. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, 1987.

[Bok88] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, Jan. 1988.

[Bra79] W. C. Brantley. *Automatically Decomposing Signal Processing Applications on Multiprocessors.* PhD thesis, Carnegie-Mellon University, 1979.

[BS84] T. Barnwell III and D. Schwarz. Optimal implementation of flow graphs on synchronous multiprocessors. In *Int. Conf. on ASSP 84*, 1984.

[CD73] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

[Cen88] Center for Digital Systems Research, Research Triangle Institute, Research Triangle Park, North Carolina 27709. *ADAS: An Architecture Design and Assessment System*, 1988. Training Manual.

[CHLE80] W. W. Chu, L. J. Hollaway, M.-T. Lan, and K. Efe. Task allocation in distributed data processing. *Computer*, 13(11):57–69, Nov. 1980.

[Chv83] V. Chvatal. *Linear Programming.* W. H. Freeman and Company, New York/San Francisco, 1983.

[CL87] W. W. Chu and L. M-T. Lan. Task allocation and precedence relations for distributed real-time systems. *IEEE Transactions on Computers*, C-36(6):667–679, June 1987.

[CS83] P. Cappello and K. Steiglitz. Unifying VLSI Array Designs with Geometric Transformations. In *Proceedings 1983 International Conference on Parallel Processing*, pages 448–457. IEEE, 1983.

[Efe82] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *Computer*, 15(6):50–56, June 1982.

[EFRV86] G. Estrin, R. Fenchel, R. Razouk, and M. Vernon. SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[ERL90] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[Est78] G. Estrin. A methodology for design of digital systems - supported by SARA at the age of one. In *Proceedings of National Computer Conference*, volume 47, pages 313–324. NCC, 1978.

[FB73] E. B. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Transactions on Computers*, C-22(8):745–751, Aug. 1973.

[FC90] R. Freund and D. Conwell. Superconcurrency: A Form of Distributed Heterogeneous Supercomputing. *Supercomputing Review*, October 1990.

[FM85] J. A. B. Fortes and D. I. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2:277–301, May 1985.

[For83] J. A. B. Fortes. *Algorithm Transformations for Parallel Processing and VLSI Architecture Design.* PhD thesis, Department of Electrical Engineering, University of Southern California, December 1983.

[GGJ78] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, Jan-Feb 1978.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, California, 1979.

[GN72] R. Garfinkel and G. Nemhauser. *Integer Programming.* John Wiley & Sons, New York, 1972.

[Gra69] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, March 1969.

[Gus83] D. Gusfield. Parametric Combinatorial Computing and a Problem of Program Module Distribution. *Journal of the Association for Computing Machinery*, 30(3):551–563, July 1983.

[GVY90] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. In *Proceedings 17th Annual International Symposium on Computer Architecture.* ACM, 1990.

[Had89a] E. K. Haddad. Analysis, modeling and optimization of multiprocessing execution time. Tech. Rep. TR 89-11, Department of Computer Science, Virginia Polytechnic Institute and State University, Falls Church, VA, 1989.

[Had89b] E. K. Haddad. Optimal Load Allocation for Parallel and Distributed Processing. Technical Report TR 89-12, Department of Computer Science, Virginia Polytechnic Institute and State University, April 1989.

[Had89c] E. K. Haddad. Partitioned load allocation for minimum parallel processing time. In *Proceedings 1989 International Conference on Parallel Processing*. IEEE Computer Society, Aug. 1989.

[Haf81] L. Hafer. *Automated Data-Memory Synthesis : A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.* PhD thesis, Dept of Electrical Engineering, Carnegie Mellon University, Pittsburgh, Pa., May 1981.

[Haf91] L. J. Hafer. Bonsai: Teaching a Clown to Prune Trees. Tech. Rep. CMPT TR 91-6, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, July 1991.

[HCAL89] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, Apr. 1989.

[HE89] B. S. Haroun and M. I. Elmasry. Architectural synthesis for DSP silicon compilers. *IEEE Trans. CAD.*, 8(4), April 1989.

[HH90] L. J. Hafer and E. Hutchings. Bringing up Bozo. Tech. Rep. CMPT TR 90-2, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, Mar. 1990.

[HHL90] C. Hwang, Y. Hsu, and Y. Lin. Optimum and Heuristic Data Path Scheduling Under Resource Constraints. In *Proceedings 27th Design Automation Conference*, pages 65–70. ACM/IEEE, June 1990.

[HLH91] Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design*, 10(4):464–475, Apr. 1991.

[Hou90] C. E. Houstis. Module allocation of real-time applications to distributed systems. *IEEE Transactions on Software Engineering*, 16(7):699–709, July 1990.

[HP83] L. J. Hafer and A. C. Parker. A formal method for the specification, analysis, and design of register-transfer level digital logic. *IEEE Transactions on Computer-Aided Design*, CAD-2(1):4–17, Jan. 1983.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, Nov. 1961.

[ISB86] M. Iqbal, J. Saltz, and S. Bokhari. A Comparative Analysis of Static and Dynamic Load Balancing Strategies. In *Proceedings 1986 International Conference on Parallel Processing*, pages 1040–1047. IEEE, August 1986.

[ISXC86] B. Indurkhya, H. S. Stone, and L. Xi-Cheng. Optimal partitioning of randomly generated distributed programs. *IEEE Transactions on Software Engineering*, SE-12(3):483–495, Mar. 1986.

[JC81] L. Johnsson and D. Cohen. A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks. In *Proceedings CMU Conf. VLSI Syst. Computations*, pages 213–225. Comput. Sci. Press, Rockville, Md., October 1981.

[JKMP89] R. Jain, K. Küçükçakar, M. Mlinar, and A. Parker. Experience with the ADAM Synthesis System. In *Proceedings 26th Design Automation Conference*, pages 56–61. ACM/IEEE, June 1989.

[JMP88] R. Jain, M. Mlinar, and A. Parker. Area-Time Model for Synthesis of Non-Pipelined Designs. In *Proceedings International Conference on Computer-Aided Design*, pages 48–51. ACM/IEEE, November 1988.

[JPP87] R. Jain, A. Parker, and N. Park. Predicting Area-Time Tradeoffs for Pipelined Design. In *Proceedings 24th Design Automation Conference*, pages 35–41. ACM/IEEE, July 1987.

[KAGER82] S. Y. Kung, K. Arun, R. Gal-Ezer, and D. Rao. Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Transactions on Computers*, C-31(11):1054–1066, November 1982.

[KL80] H. T. Kung and C. E. Leiserson. Systolic Arrays for VLSI. In *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980. Section 8.3.

[KL88] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, pages 23–32, January 1988.

[KN84] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, Nov. 1984.

[Kuc91] K. Küçükçakar. *System-Level Synthesis Techniques with Emphasis on Partitioning and Design Planning.* PhD thesis, University of Southern California, Septemeber 1991.

[Kun79] H. T. Kung. Let's Design Algorithms for VLSI Systems. In *Proceedings Caltech Conf. VLSI*, pages 65–90, January 1979.

[Kun82] H. T. Kung. Why Systolic Architectures. *Computer*, 15(1):37–46, January 1982.

[Kun84] Sun-Yuan Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 72(7):867–884, July 1984.

[Law78] G. Lawson. Design style selector, an automated computer program implementation. Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., August 1978.

[LB90] E. A. Lee and J. C. Bier. Architectures for Statically Scheduled Dataflow. *Journal of Parallel and Distributed Computing*, 10:333–348, 1990.

[LW85] Guo-Jie Li and B. W. Wah. The design of optimal systolic arrays. *IEEE Transactions on Computers*, C-34(1):66–77, Jan. 1985.

[McD82] J. McDermott. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence*, 19(2), 1982.

[MLT82] Perng-Yi R. Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, Jan. 1982.

[MM78] R. Marsten and T. Morin. A Hybrid Approach to Discrete Mathematical Programming. *Mathematical Programming*, 14(1):21–40, January 1978.

[Mol82a] D. I. Moldovan. Computational Models for VLSI Systems. Technical Report DIM-82-3, Department of Electrical Engineering, University of Southern California, 1982.

[Mol82b] D. I. Moldovan. On the Analysis and Synthesis of VLSI Algorithms. *IEEE Transactions on Computers*, C-31(11):1121–1126, November 1982.

[Mol83] D. I. Moldovan. On the Design of Algorithms for VLSI Systolic Arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[MPC90] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, Feb. 1990.

[MT82] R. Mehrotra and S. N. Talukdar. Task scheduling on multiprocessors. Tech. Rep. DRC-18-55-82, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, PA, Dec. 1982.

[MT84] R. Mehrotra and S. N. Talukdar. Scheduling of tasks for distributed processors. Tech. Rep. DRC-18-68-84, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, PA, Dec. 1984.

[MW84] W. L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.

[Nad86] B. A. Nadel. *The Consistent Labeling Problem and Its Algorithms: Towards Exact-Case Complexities and Theory-Based Heuristics.* PhD thesis, Computer Science Department, Rutgers University, New Brunswick, N.J., May 1986.

[Nad89] B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5:188–224, November 1989.

[Nic89] D. M. Nicol. Optimal partitioning of random programs across two processors. *IEEE Transactions on Software Engineering*, 15(2), Feb. 1989.

[Nil71] N. J. Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[Par85] N. Park. *Synthesis of High-Speed Digital Systems.* PhD thesis, University of Southern California, August 1985.

[PHJ+88] A. C. Parker, S. Hayati, R. Jain, K. Küçükçakar, M. Mlinar, S. Prakash, and J. Siedel. High-Level Synthesis in the ADAM System. In *2nd International Workshop of VLSI Design, Bangalore, India*, December 1988.

[PKPW91] A. C. Parker, K. Küçükçakar, S. Prakash, and J.-P. Weng. Unified System Construction (USC). In R. Camposano and W. Wolf, editors, *High-level VLSI Synthesis*, chapter 14, pages 331–354. Kluwer Academic Publishers, Boston, 1991.

[PP82] C. Price and U. Pooch. Search Techniques for a Nonlinear Multiprocessor Scheduling Problem. *Naval Research Logistics Quarterly*, 29(2):213–233, June 1982.

[PP86] N. Park and A. Parker. Sehwa:A Program for Synthesis of Pipelines. In *Proceedings 23rd Design Automation Conference*, pages 454–460. ACM/IEEE, July 1986.

[PP90] S. Prakash and A. C. Parker. A Formal Model for Representation of Multiprocessor Systems. IEEE Design Automation Workshop, January 1990.

[PP91a] S. Prakash and A. C. Parker. Synthesis of Application-Specific Multiprocessor Architectures. In *Proceedings 28th Design Automation Conference*, pages 8–13. ACM/IEEE, June 1991.

[PP91b] S. Prakash and A. C. Parker. Synthesis of Application-Specific Multiprocessor Architectures. In *Fifth International Workshop on High-Level Synthesis*. ACM, March 1991.

[PP92a] S. Prakash and A. C. Parker. A Design Method for Optimal Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. In *Proceedings IPPS '92 - Workshop on Heterogeneous Processing*. ACM/IEEE, March 1992.

[PP92b] S. Prakash and A. C. Parker. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 16:338-351, December 1992.

[PP92c] S. Prakash and A. C. Parker. Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. In *Proceedings 19th Int'l Symp. on Computer Architecture*. IEEE, ACM SIGArch, May 1992. Presented as a poster.

[PP92d] S. Prakash and A. C. Parker. Synthesis of Application-Specific Multiprocessor Systems Including Memory Components. In *Proceedings Int'l Conf. on Application-Specific Array Processors*, pages 118–132. IEEE Computer Society, Aug. 1992. Highlight Talk (Invited paper).

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proceedings 23rd Design Automation Conference*, pages 461–466. ACM/IEEE, July 1986.

[Pra87] S. Prakash. Guiding design decisions in RT-level logic synthesis. Master's thesis, School of Computing Science, Simon Fraser University, Burnaby, B.C., Apr. 1987.

[RM87] Jan Rabaey and Hugo De Man. Computer aided design of digital signal processing systems. In *Proc. ICCD*, 1987.

[RSH79] G. S. Rao, H. S. Stone, and T. C. Hu. Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers*, C-28(4):291–299, Apr. 1979.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, 1989.

[SB78] H. S. Stone and S. H. Bokhari. Control of Distributed Processes. *Computer*, 11:97–106, July 1978.

[Sid90] G. Sidebottom. Satisfaction of Non-Negative Integer Arithmetic Expressions. Open File Report 1990-15, Alberta Research Council, Calgary, Alberta, T2E 7H7, July 1990.

[SSKD87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV. An overview of the PASM parallel processing system. In D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, editors, *Computer Architecture*, pages 387–407. IEEE Computer Society Press, Washington, D.C., 1987.

[ST85] Chien-Chung Shen and Wen-Hsiang Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, Mar. 1985.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, Jan. 1977.

[Sto78] H. S. Stone. Critical Load Factors in Two-processor Distributed Systems. *IEEE Transactions on Software Engineering*, SE-4:254–258, May 1978.

[Sto87] H. S. Stone. *High Performance Computer Architecture.* Addison-Wesley, 1987.

[Tho77] D. Thomas. *The Design and Analysis of an Automated Design Style Selector.* PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April 1977.

[WD81] U. Weiser and A. Davis. A Wavefront Notation Tool for VLSI Array Design. In *Proceedings CMU Conf. VLSI Syst. Computations*, pages 226–234. Comput. Sci. Press, Rockville, Md., October 1981.

[XMP88] XMP Software, Inc., Tucson, AZ 85732. *XML2 Modeling Language*, 1988. Tutorial.

Appendix A

The MILP Model for Four-Subtask Task Graph

In this appendix, we present the complete MILP model for the four-subtask task graph (the model for nine-subtask task graph is too long to be included here). The model is written in the *XML2 Modeling Language*, the syntax of which is described in [XMP88]. Bozo accepts input in this format. *PARAMETERS* in the model list the constant parameter values; *VARIABLES* list the real variables (or the timing variables in our case); *VARIABLES BINARY* list the 0-1 variables; *EQUATIONS* list the constraints and the objective function (the equation SYSTIM is the objective function in this case).

A.1 The Model

```
$ FOUR-NODE-TASK-GRAPH MODEL

$  PARAMETERS

! Constant parameter values used in the model

   @TAV_11 = 100
   @TAV_21 = 100
   @TAV_42 = 100

   @TP_11  = 1
   @TP_12  = 3
   @TP_21  = 1
   @TP_22  = 1
   @TP_23  = 3
   @TP_32  = 2
```

```
@TP_33  = 1
@TP_41  = 3
@TP_42  = 1

@DCR    = 1
@V_31   = 1
@V_32   = 1
@V_41   = 1
@D_31   = 1
@D_32   = 1
@D_41   = 1

@TMAX   = 15
@TMAX2  = 30
@TMAX3  = 45
@TMAX4  = 60
@TMAX5  = 75

@COST_1 = 40
@COST_2 = 50
@COST_3 = 20

@C_LINK = 10

@CMAX   = 220

@FI_11  = 0.25
@FI_21  = 0.25
@FI_31  = 0.25
@FI_32  = 0.50
@FI_41  = 0.25
@FI_42  = 0.50

@FI_111 = 0.75
@FI_211 = 0.75
@FI_311 = 0.75
@FI_321 = 0.50
@FI_411 = 0.75
@FI_421 = 0.50

@FO_11  = 0.50
```

```
   @FO_12  = 0.75
   @FO_21  = 0.50
   @FO_22  = 0.75
   @FO_31  = 0.75
   @FO_41  = 0.75

   @FO_111 = 0.50
   @FO_121 = 0.25
   @FO_211 = 0.50
   @FO_221 = 0.25
   @FO_311 = 0.25
   @FO_411 = 0.25

$  VARIABLES

! Real (timing) variables in the model

   TSS_1, TSS_2, TSS_3, TSS_4
   TSE_1, TSE_2, TSE_3, TSE_4
   TOA_11, TOA_12, TOA_21, TOA_22, TOA_31, TOA_41
   TCS_31, TCS_32, TCS_41
   TCE_31, TCE_32, TCE_41
   TSYS

$  VARIABLES BINARY

! 0-1 variables in the model

   SIG_1A1, SIG_2A1
   SIG_1A2, SIG_1B2, SIG_2A2, SIG_2B2, SIG_3A2
   SIG_2A3, SIG_2B3, SIG_2C3, SIG_3A3, SIG_3B3
   SIG_1A4, SIG_1B4, SIG_1C4, SIG_2A4, SIG_2B4, SIG_2C4, SIG_2D4

   GAM_13, GAM_23, GAM_14

   BET_1A, BET_1B, BET_1C, BET_2A, BET_2B, BET_2C, BET_2D, BET_3A,
   BET_3B

   CHI_1A1B, CHI_1A1C, CHI_1A2A, CHI_1A2B, CHI_1A2C, CHI_1A2D,
   CHI_1A3A, CHI_1A3B
   CHI_1B2A, CHI_1B2B, CHI_1B2C, CHI_1B3A, CHI_1B3B
```

```
   CHI_2A1A, CHI_2A1B, CHI_2A1C, CHI_2A2B, CHI_2A2C, CHI_2A2D,
   CHI_2A3A, CHI_2A3B
   CHI_2B2A, CHI_2B2C, CHI_2B3A, CHI_2B3B
   CHI_3A2A, CHI_3A2B, CHI_3A2C, CHI_3A3B

   DEL_2A13, DEL_1A14, DEL_2A14, DEL_2A23, DEL_2B23, DEL_3A23

   ALF_12, ALF_24, ALF_34

   PHI_3132, PHI_3141, PHI_3241

$  EQUATIONS

! Constraints in the model

   MAPTSK1  ..  SIG_1A1 + SIG_2A1  =E=  1
   MAPTSK2  ..  SIG_1A2 + SIG_1B2 + SIG_2A2 + SIG_2B2 + SIG_3A2
                =E=  1
   MAPTSK3  ..  SIG_2A3 + SIG_2B3 + SIG_2C3 + SIG_3A3 + SIG_3B3
                =E=  1
   MAPTSK4  ..  SIG_1A4 + SIG_1B4 + SIG_1C4
              + SIG_2A4 + SIG_2B4 + SIG_2C4 + SIG_2D4  =E=  1

   DTTYP_13  ..  GAM_13 + DEL_2A13  =E=  1
   DDL2A131 ..  DEL_2A13 - SIG_2A1  =L=  0
   DDL2A133 ..  DEL_2A13 - SIG_2A3  =L=  0

   DTTYP_14  ..  GAM_14 + DEL_1A14 + DEL_2A14  =E=  1
   DDL1A141 ..  DEL_1A14 - SIG_1A1  =L=  0
   DDL1A144 ..  DEL_1A14 - SIG_1A4  =L=  0
   DDL2A141 ..  DEL_2A14 - SIG_2A1  =L=  0
   DDL2A144 ..  DEL_2A14 - SIG_2A4  =L=  0

   DDTYP_23  ..  GAM_23 + DEL_2A23 + DEL_2B23 + DEL_3A23  =E=  1
   DDL2A232 ..  DEL_2A23 - SIG_2A2  =L=  0
   DDL2A233 ..  DEL_2A23 - SIG_2A3  =L=  0
   DDL2B232 ..  DEL_2B23 - SIG_2B2  =L=  0
   DDL2B233 ..  DEL_2B23 - SIG_2B3  =L=  0
   DDL3A232 ..  DEL_3A23 - SIG_3A2  =L=  0
   DDL3A233 ..  DEL_3A23 - SIG_3A3  =L=  0
```

```
BEGTSK1  ..  @FI_111 * TSS_1 + @FI_11 * TSE_1  =G=  @TAV_11
BEGTSK2  ..  @FI_211 * TSS_2 + @FI_21 * TSE_2  =G=  @TAV_21
BEGTSK31 ..  @FI_311 * TSS_3 + @FI_31 * TSE_3 - TCE_31  =G=  0
BEGTSK32 ..  @FI_321 * TSS_3 + @FI_32 * TSE_3 - TCE_32  =G=  0
BEGTSK41 ..  @FI_411 * TSS_4 + @FI_41 * TSE_4 - TCE_41  =G=  0
BEGTSK42 ..  @FI_421 * TSS_4 + @FI_42 * TSE_4  =G=  @TAV_42

ENDTSK1  ..  TSE_1 - TSS_1 - @TP_11 * SIG_1A1 - @TP_12 * SIG_2A1
             =E=  0
ENDTSK2  ..  TSE_2 - TSS_2 - @TP_21 * SIG_1A2 - @TP_21 * SIG_1B2
            - @TP_22 * SIG_2A2 - @TP_22 * SIG_2B2
            - @TP_23 * SIG_3A2  =E=  0
ENDTSK3  ..  TSE_3 - TSS_3 - @TP_32 * SIG_2A3 - @TP_32 * SIG_2B3
            - @TP_32 * SIG_2C3 - @TP_33 * SIG_3A3
            - @TP_33 * SIG_3B3  =E=  0
ENDTSK4  ..  TSE_4 - TSS_4 - @TP_41 * SIG_1A4 - @TP_41 * SIG_1B4
            - @TP_41 * SIG_1C4 - @TP_42 * SIG_2A4
            - @TP_42 * SIG_2B4 - @TP_42 * SIG_2C4
            - @TP_42 * SIG_2D4  =E=  0

OUTPUT11 ..  TOA_11 - @FO_111 * TSS_1 - @FO_11 * TSE_1  =E=  0
OUTPUT12 ..  TOA_12 - @FO_121 * TSS_1 - @FO_12 * TSE_1  =E=  0
OUTPUT21 ..  TOA_21 - @FO_211 * TSS_2 - @FO_21 * TSE_2  =E=  0
OUTPUT22 ..  TOA_22 - @FO_221 * TSS_2 - @FO_22 * TSE_2  =E=  0
OUTPUT31 ..  TOA_31 - @FO_311 * TSS_3 - @FO_31 * TSE_3  =E=  0
OUTPUT41 ..  TOA_41 - @FO_411 * TSS_4 - @FO_41 * TSE_4  =E=  0

SYST11    ..  TSYS - TSE_1 + TSS_1  =G=  0
SYST12    ..  TSYS - TSE_1 + TSS_2  =G=  0
SYST13    ..  TSYS - TSE_1 + TSS_3  =G=  0
SYST14    ..  TSYS - TSE_1 + TSS_4  =G=  0
SYST21    ..  TSYS - TSE_2 + TSS_1  =G=  0
SYST22    ..  TSYS - TSE_2 + TSS_2  =G=  0
SYST23    ..  TSYS - TSE_2 + TSS_3  =G=  0
SYST24    ..  TSYS - TSE_2 + TSS_4  =G=  0
SYST31    ..  TSYS - TSE_3 + TSS_1  =G=  0
SYST32    ..  TSYS - TSE_3 + TSS_2  =G=  0
SYST33    ..  TSYS - TSE_3 + TSS_3  =G=  0
SYST34    ..  TSYS - TSE_3 + TSS_4  =G=  0
SYST41    ..  TSYS - TSE_4 + TSS_1  =G=  0
SYST42    ..  TSYS - TSE_4 + TSS_2  =G=  0
```

```
SYST43     ..  TSYS - TSE_4 + TSS_3  =G=  0
SYST44     ..  TSYS - TSE_4 + TSS_4  =G=  0

BEGCOM31 ..  TCS_31 - TOA_11  =G=  0
BEGCOM32 ..  TCS_32 - TOA_21  =G=  0
BEGCOM41 ..  TCS_41 - TOA_12  =G=  0

ENDCOM31 ..  TCE_31 - TCS_31 - @D_31 * GAM_13  =E=  0
ENDCOM32 ..  TCE_32 - TCS_32 - @D_32 * GAM_23  =E=  0
ENDCOM41 ..  TCE_41 - TCS_41 - @D_41 * GAM_14  =E=  0

PLP1A121 ..  TSS_2 - TSE_1 - @TMAX * ALF_12 - @TMAX * SIG_1A1
            - @TMAX * SIG_1A2  =G=  - @TMAX3
PLP1A120 ..  TSS_1 - TSE_2 + @TMAX * ALF_12 - @TMAX * SIG_1A1
            - @TMAX * SIG_1A2  =G=  - @TMAX2
PLP2A121 ..  TSS_2 - TSE_1 - @TMAX * ALF_12 - @TMAX * SIG_2A1
            - @TMAX * SIG_2A2  =G=  - @TMAX3
PLP2A120 ..  TSS_1 - TSE_2 + @TMAX * ALF_12 - @TMAX * SIG_2A1
            - @TMAX * SIG_2A2  =G=  - @TMAX2
PLP2A13  ..  TSS_3 - TSE_1 - @TMAX * SIG_2A1 - @TMAX * SIG_2A3
            =G=  - @TMAX2
PLP1A14  ..  TSS_4 - TSE_1 - @TMAX * SIG_1A1 - @TMAX * SIG_1A4
            =G=  - @TMAX2
PLP2A14  ..  TSS_4 - TSE_1 - @TMAX * SIG_2A1 - @TMAX * SIG_2A4
            =G=  - @TMAX2
PLP2A23  ..  TSS_3 - TSE_2 - @TMAX * SIG_2A2 - @TMAX * SIG_2A3
            =G=  - @TMAX2
PLP2B23  ..  TSS_3 - TSE_2 - @TMAX * SIG_2B2 - @TMAX * SIG_2B3
            =G=  - @TMAX2
PLP3A23  ..  TSS_3 - TSE_2 - @TMAX * SIG_3A2 - @TMAX * SIG_3A3
            =G=  - @TMAX2
PLP1A241 ..  TSS_4 - TSE_2 - @TMAX * ALF_24 - @TMAX * SIG_1A2
            - @TMAX * SIG_1A4  =G=  - @TMAX3
PLP1A240 ..  TSS_2 - TSE_4 + @TMAX * ALF_24 - @TMAX * SIG_1A2
            - @TMAX * SIG_1A4  =G=  - @TMAX2
PLP1B241 ..  TSS_4 - TSE_2 - @TMAX * ALF_24 - @TMAX * SIG_1B2
            - @TMAX * SIG_1B4  =G=  - @TMAX3
PLP1B240 ..  TSS_2 - TSE_4 + @TMAX * ALF_24 - @TMAX * SIG_1B2
            - @TMAX * SIG_1B4  =G=  - @TMAX2
PLP2A241 ..  TSS_4 - TSE_2 - @TMAX * ALF_24 - @TMAX * SIG_2A2
            - @TMAX * SIG_2A4  =G=  - @TMAX3
```

```
PLP2A240 ..  TSS_2 - TSE_4 + @TMAX * ALF_24 - @TMAX * SIG_2A2
            - @TMAX * SIG_2A4  =G=  - @TMAX2
PLP2B241 ..  TSS_4 - TSE_2 - @TMAX * ALF_24 - @TMAX * SIG_2B2
            - @TMAX * SIG_2B4  =G=  - @TMAX3
PLP2B240 ..  TSS_2 - TSE_4 + @TMAX * ALF_24 - @TMAX * SIG_2B2
            - @TMAX * SIG_2B4  =G=  - @TMAX2
PLP2A341 ..  TSS_4 - TSE_3 - @TMAX * ALF_34 - @TMAX * SIG_2A3
            - @TMAX * SIG_2A4  =G=  - @TMAX3
PLP2A340 ..  TSS_3 - TSE_4 + @TMAX * ALF_34 - @TMAX * SIG_2A3
            - @TMAX * SIG_2A4  =G=  - @TMAX2
PLP2B341 ..  TSS_4 - TSE_3 - @TMAX * ALF_34 - @TMAX * SIG_2B3
            - @TMAX * SIG_2B4  =G=  - @TMAX3
PLP2B340 ..  TSS_3 - TSE_4 + @TMAX * ALF_34 - @TMAX * SIG_2B3
            - @TMAX * SIG_2B4  =G=  - @TMAX2
PLP2C341 ..  TSS_4 - TSE_3 - @TMAX * ALF_34 - @TMAX * SIG_2C3
            - @TMAX * SIG_2C4  =G=  - @TMAX3
PLP2C340 ..  TSS_3 - TSE_4 + @TMAX * ALF_34 - @TMAX * SIG_2C3
            - @TMAX * SIG_2C4  =G=  - @TMAX2

CLP31321 ..  TCS_32 - TCE_31 - @TMAX * PHI_3132 - @TMAX * SIG_1A1
            - @TMAX * SIG_1A2  =G=  - @TMAX3
CLP31322 ..  TCS_31 - TCE_32 + @TMAX * PHI_3132 - @TMAX * SIG_1A1
            - @TMAX * SIG_1A2  =G=  - @TMAX2
CLP31323 ..  TCS_32 - TCE_31 - @TMAX * PHI_3132 - @TMAX * SIG_2A1
            - @TMAX * SIG_2A2  =G=  - @TMAX3
CLP31324 ..  TCS_31 - TCE_32 + @TMAX * PHI_3132 - @TMAX * SIG_2A1
            - @TMAX * SIG_2A2  =G=  - @TMAX2
CLP31411 ..  TCS_41 - TCE_31 - @TMAX * PHI_3141 - @TMAX * SIG_2A3
            - @TMAX * SIG_2A4  =G=  - @TMAX3
CLP31412 ..  TCS_31 - TCE_41 + @TMAX * PHI_3141 - @TMAX * SIG_2A3
            - @TMAX * SIG_2A4  =G=  - @TMAX2
CLP31413 ..  TCS_41 - TCE_31 - @TMAX * PHI_3141 - @TMAX * SIG_2B3
            - @TMAX * SIG_2B4  =G=  - @TMAX3
CLP31414 ..  TCS_31 - TCE_41 + @TMAX * PHI_3141 - @TMAX * SIG_2B3
            - @TMAX * SIG_2B4  =G=  - @TMAX2
CLP31415 ..  TCS_41 - TCE_31 - @TMAX * PHI_3141 - @TMAX * SIG_2C3
            - @TMAX * SIG_2C4  =G=  - @TMAX3
CLP31416 ..  TCS_31 - TCE_41 + @TMAX * PHI_3141 - @TMAX * SIG_2C3
            - @TMAX * SIG_2C4  =G=  - @TMAX2
CLP32411 ..  TCS_41 - TCE_32 - @TMAX * PHI_3241 - @TMAX * SIG_1A1
            - @TMAX * SIG_2A4 - @TMAX * SIG_1A2 - @TMAX * SIG_2A3
```

```
                =G=   - @TMAX5
CLP32412 ..   TCS_32 - TCE_41 + @TMAX * PHI_3241 - @TMAX * SIG_1A1
             - @TMAX * SIG_2A4 - @TMAX * SIG_1A2 - @TMAX * SIG_2A3
                =G=   - @TMAX4
CLP32413 ..   TCS_41 - TCE_32 - @TMAX * PHI_3241 - @TMAX * SIG_1A1
             - @TMAX * SIG_2B4 - @TMAX * SIG_1A2 - @TMAX * SIG_2B3
                =G=   - @TMAX5
CLP32414 ..   TCS_32 - TCE_41 + @TMAX * PHI_3241 - @TMAX * SIG_1A1
             - @TMAX * SIG_2B4 - @TMAX * SIG_1A2 - @TMAX * SIG_2B3
                =G=   - @TMAX4
CLP32415 ..   TCS_41 - TCE_32 - @TMAX * PHI_3241 - @TMAX * SIG_1A1
             - @TMAX * SIG_2C4 - @TMAX * SIG_1A2 - @TMAX * SIG_2C3
                =G=   - @TMAX5
CLP32416 ..   TCS_32 - TCE_41 + @TMAX * PHI_3241 - @TMAX * SIG_1A1
             - @TMAX * SIG_2C4 - @TMAX * SIG_1A2 - @TMAX * SIG_2C3
                =G=   - @TMAX4
CLP32417 ..   TCS_41 - TCE_32 - @TMAX * PHI_3241 - @TMAX * SIG_2A1
             - @TMAX * SIG_2B4 - @TMAX * SIG_2A2 - @TMAX * SIG_2B3
                =G=   - @TMAX5
CLP32418 ..   TCS_32 - TCE_41 + @TMAX * PHI_3241 - @TMAX * SIG_2A1
             - @TMAX * SIG_2B4 - @TMAX * SIG_2A2 - @TMAX * SIG_2B3
                =G=   - @TMAX4
CLP32419 ..   TCS_41 - TCE_32 - @TMAX * PHI_3241 - @TMAX * SIG_2A1
             - @TMAX * SIG_2C4 - @TMAX * SIG_2A2 - @TMAX * SIG_2C3
                =G=   - @TMAX5
CLP3241A ..   TCS_32 - TCE_41 + @TMAX * PHI_3241 - @TMAX * SIG_2A1
             - @TMAX * SIG_2C4 - @TMAX * SIG_2A2 - @TMAX * SIG_2C3
                =G=   - @TMAX4

MAP_1A1    ..   BET_1A - SIG_1A1   =G=   0
MAP_1A2    ..   BET_1A - SIG_1A2   =G=   0
MAP_1A4    ..   BET_1A - SIG_1A4   =G=   0
MAP_1B2    ..   BET_1B - SIG_1B2   =G=   0
MAP_1B4    ..   BET_1B - SIG_1B4   =G=   0
MAP_1C4    ..   BET_1C - SIG_1C4   =E=   0
MAP_2A1    ..   BET_2A - SIG_2A1   =G=   0
MAP_2A2    ..   BET_2A - SIG_2A2   =G=   0
MAP_2A3    ..   BET_2A - SIG_2A3   =G=   0
MAP_2A4    ..   BET_2A - SIG_2A4   =G=   0
MAP_2B2    ..   BET_2B - SIG_2B2   =G=   0
MAP_2B3    ..   BET_2B - SIG_2B3   =G=   0
```

```
MAP_2B4    ..  BET_2B - SIG_2B4  =G=  0
MAP_2C3    ..  BET_2C - SIG_2C3  =G=  0
MAP_2C4    ..  BET_2C - SIG_2C4  =G=  0
MAP_2D4    ..  BET_2D - SIG_2D4  =E=  0
MAP_3A2    ..  BET_3A - SIG_3A2  =G=  0
MAP_3A3    ..  BET_3A - SIG_3A3  =G=  0
MAP_3B3    ..  BET_3B - SIG_3B3  =E=  0

LN1A1B14 ..  CHI_1A1B - SIG_1A1 - SIG_1B4  =G=  - 1
LN1A1C14 ..  CHI_1A1C - SIG_1A1 - SIG_1C4  =G=  - 1
LN1A2A13 ..  CHI_1A2A - SIG_1A1 - SIG_2A3  =G=  - 1
LN1A2A14 ..  CHI_1A2A - SIG_1A1 - SIG_2A4  =G=  - 1
LN1A2A23 ..  CHI_1A2A - SIG_1A2 - SIG_2A3  =G=  - 1
LN1A2B13 ..  CHI_1A2B - SIG_1A1 - SIG_2B3  =G=  - 1
LN1A2B14 ..  CHI_1A2B - SIG_1A1 - SIG_2B4  =G=  - 1
LN1A2B23 ..  CHI_1A2B - SIG_1A2 - SIG_2B3  =G=  - 1
LN1A2C13 ..  CHI_1A2C - SIG_1A1 - SIG_2C3  =G=  - 1
LN1A2C14 ..  CHI_1A2C - SIG_1A1 - SIG_2C4  =G=  - 1
LN1A2C23 ..  CHI_1A2C - SIG_1A2 - SIG_2C3  =G=  - 1
LN1A2D14 ..  CHI_1A2D - SIG_1A1 - SIG_2D4  =G=  - 1
LN1A3A13 ..  CHI_1A3A - SIG_1A1 - SIG_3A3  =G=  - 1
LN1A3A23 ..  CHI_1A3A - SIG_1A2 - SIG_3A3  =G=  - 1
LN1A3B13 ..  CHI_1A3B - SIG_1A1 - SIG_3B3  =G=  - 1
LN1A3B23 ..  CHI_1A3B - SIG_1A2 - SIG_3B3  =G=  - 1
LN1B2A23 ..  CHI_1B2A - SIG_1B2 - SIG_2A3  =G=  - 1
LN1B2B23 ..  CHI_1B2B - SIG_1B2 - SIG_2B3  =G=  - 1
LN1B2C23 ..  CHI_1B2C - SIG_1B2 - SIG_2C3  =G=  - 1
LN1B3A23 ..  CHI_1B3A - SIG_1B2 - SIG_3A3  =G=  - 1
LN1B3B23 ..  CHI_1B3B - SIG_1B2 - SIG_3B3  =G=  - 1
LN2A1A14 ..  CHI_2A1A - SIG_2A1 - SIG_1A4  =G=  - 1
LN2A1B14 ..  CHI_2A1B - SIG_2A1 - SIG_1B4  =G=  - 1
LN2A1C14 ..  CHI_2A1C - SIG_2A1 - SIG_1C4  =G=  - 1
LN2A2B13 ..  CHI_2A2B - SIG_2A1 - SIG_2B3  =G=  - 1
LN2A2B14 ..  CHI_2A2B - SIG_2A1 - SIG_2B4  =G=  - 1
LN2A2B23 ..  CHI_2A2B - SIG_2A2 - SIG_2B3  =G=  - 1
LN2A2C13 ..  CHI_2A2C - SIG_2A1 - SIG_2C3  =G=  - 1
LN2A2C14 ..  CHI_2A2C - SIG_2A1 - SIG_2C4  =G=  - 1
LN2A2C23 ..  CHI_2A2C - SIG_2A2 - SIG_2C3  =G=  - 1
LN2A2D14 ..  CHI_2A2D - SIG_2A1 - SIG_2D4  =G=  - 1
LN2A3A13 ..  CHI_2A3A - SIG_2A1 - SIG_3A3  =G=  - 1
LN2A3A23 ..  CHI_2A3A - SIG_2A2 - SIG_3A3  =G=  - 1
```

```
LN2A3B13 ..  CHI_2A3B - SIG_2A1 - SIG_3B3  =G=  - 1
LN2A3B23 ..  CHI_2A3B - SIG_2A2 - SIG_3B3  =G=  - 1
LN2B2A23 ..  CHI_2B2A - SIG_2B2 - SIG_2A3  =G=  - 1
LN2B2C23 ..  CHI_2B2C - SIG_2B2 - SIG_2C3  =G=  - 1
LN2B3A23 ..  CHI_2B3A - SIG_2B2 - SIG_3A3  =G=  - 1
LN2B3B23 ..  CHI_2B3B - SIG_2B2 - SIG_3B3  =G=  - 1
LN3A2A23 ..  CHI_3A2A - SIG_3A2 - SIG_2A3  =G=  - 1
LN3A2B23 ..  CHI_3A2B - SIG_3A2 - SIG_2B3  =G=  - 1
LN3A2C23 ..  CHI_3A2C - SIG_3A2 - SIG_2C3  =G=  - 1
LN3A3B23 ..  CHI_3A3B - SIG_3A2 - SIG_3B3  =G=  - 1

MAP1A1B   ..  BET_1A - BET_1B  =G=  0
MAP1A1C   ..  BET_1A - BET_1C  =G=  0
MAP1B1C   ..  BET_1B - BET_1C  =G=  0
MAP2A2B   ..  BET_2A - BET_2B  =G=  0
MAP2A2C   ..  BET_2A - BET_2C  =G=  0
MAP2A2D   ..  BET_2A - BET_2D  =G=  0
MAP2B2C   ..  BET_2B - BET_2C  =G=  0
MAP2B2D   ..  BET_2B - BET_2D  =G=  0
MAP2C2D   ..  BET_2C - BET_2D  =G=  0
MAP3A3B   ..  BET_3A - BET_3B  =G=  0

SYSCOST  ..  @COST_1 * BET_1A + @COST_1 * BET_1B
           + @COST_1 * BET_1C + @COST_2 * BET_2A
           + @COST_2 * BET_2B + @COST_2 * BET_2C
           + @COST_2 * BET_2D + @COST_3 * BET_3A
           + @COST_3 * BET_3B
           + @C_LINK * CHI_1A1B + @C_LINK * CHI_1A1C
           + @C_LINK * CHI_1A2A + @C_LINK * CHI_1A2B
           + @C_LINK * CHI_1A2C + @C_LINK * CHI_1A2D
           + @C_LINK * CHI_1A3A + @C_LINK * CHI_1A3B
           + @C_LINK * CHI_1B2A + @C_LINK * CHI_1B2B
           + @C_LINK * CHI_1B2C + @C_LINK * CHI_1B3A
           + @C_LINK * CHI_1B3B + @C_LINK * CHI_2A1A
           + @C_LINK * CHI_2A1B + @C_LINK * CHI_2A1C
           + @C_LINK * CHI_2A2B + @C_LINK * CHI_2A2C
           + @C_LINK * CHI_2A2D + @C_LINK * CHI_2A3A
           + @C_LINK * CHI_2A3B + @C_LINK * CHI_2B2A
           + @C_LINK * CHI_2B2C + @C_LINK * CHI_2B3A
           + @C_LINK * CHI_2B3B + @C_LINK * CHI_3A2A
           + @C_LINK * CHI_3A2B + @C_LINK * CHI_3A2C
```

```
              + @C_LINK * CHI_3A3B  =N=

! Objective function for the model

   SYSTIM    ..  1000 TSYS
              + GAM_13 + GAM_23 + GAM_14
              + @COST_1 * BET_1A + @COST_1 * BET_1B
              + @COST_1 * BET_1C + @COST_2 * BET_2A
              + @COST_2 * BET_2B + @COST_2 * BET_2C
              + @COST_2 * BET_2D + @COST_3 * BET_3A
              + @COST_3 * BET_3B
              + @C_LINK * CHI_1A1B + @C_LINK * CHI_1A1C
              + @C_LINK * CHI_1A2A + @C_LINK * CHI_1A2B
              + @C_LINK * CHI_1A2C + @C_LINK * CHI_1A2D
              + @C_LINK * CHI_1A3A + @C_LINK * CHI_1A3B
              + @C_LINK * CHI_1B2A + @C_LINK * CHI_1B2B
              + @C_LINK * CHI_1B2C + @C_LINK * CHI_1B3A
              + @C_LINK * CHI_1B3B + @C_LINK * CHI_2A1A
              + @C_LINK * CHI_2A1B + @C_LINK * CHI_2A1C
              + @C_LINK * CHI_2A2B + @C_LINK * CHI_2A2C
              + @C_LINK * CHI_2A2D + @C_LINK * CHI_2A3A
              + @C_LINK * CHI_2A3B + @C_LINK * CHI_2B2A
              + @C_LINK * CHI_2B2C + @C_LINK * CHI_2B3A
              + @C_LINK * CHI_2B3B + @C_LINK * CHI_3A2A
              + @C_LINK * CHI_3A2B + @C_LINK * CHI_3A2C
              + @C_LINK * CHI_3A3B  =N=

$  BOUNDS

$  END
```