CO-SYNTHESIS OF HARDWARE AND SOFTWARE FOR DIGITAL EMBEDDED SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Rajesh Kumar Gupta
December 10, 1993

© Copyright 1994
by
Rajesh Kumar Gupta

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Michael J. Flynn

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Krishna Saraswat

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Kunle Olukotun

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Martin Freeman, Philips Research Labs.

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

As the complexity of systems being subject to computer-aided synthesis and optimization techniques increases, so does the need to find ways to incorporate *predesigned* components into the final system implementation. In this context, a general-purpose microprocessor provides a sophisticated low-cost component that can be tailored to realize most system functions through appropriate software. This approach is particularly useful in the design of embedded systems that have a relatively simple target architecture, when compared to general-purpose computing systems such as workstations. In embedded systems the processor is used as a resource dedicated to implement specific functions. However, the design issues in embedded systems are complicated since most of these systems operate in a time-constrained environment. Recent advances in chip-level synthesis have made it possible to synthesize application-specific circuits under strict timing constraints. This dissertation formulates the problem of computer-aided design of embedded systems using both application-specific as well as general-purpose reprogrammable components under timing constraints.

Given a specification of system functionality and constraints in a hardware description language, we model the system as a set of bilogic flow graphs, and formulate the co-synthesis problem as a partitioning problem under constraints. Timing constraints are used to determine the parts of the system functionality that are delegated to application-specific hardware and the software that runs on the processor. The software component of such a 'mixed' system poses an interesting problem due to its interaction with concurrently operating hardware. We address this problem by generating software as a set of concurrent fixed-latency serialized operations called threads. The satisfaction of the imposed performance constraints is then ensured by exploiting concurrency between

program threads, achieved by an inter-leaved execution on a single processor system.

This co-synthesis of hardware and software from behavioral specifications makes it possible to build time-constrained embedded systems by using off-the-shelf parts and application-specific circuitry. Due to the reduction in size of application-specific hardware needed compared to an all-hardware solution, the needed hardware component can be easily mapped to semicustom VLSI such as gate arrays, thus shortening the design time. In addition, the ability to perform a detailed analysis of timing performance provides an opportunity to improve the system definition by creating better prototypes. The algorithms and techniques described have been implemented in a framework called *Vulcan*, which is integrated with the Stanford Olympus Synthesis System and provides a path from chip-level synthesis to system-level synthesis.

Dedication

This thesis is dedicated to a very special person in my life, my wife Anne Usha.

Acknowledgements

My deepest gratitude is to Professor Giovanni De Micheli for giving me the opportunity to explore new grounds in the computer-aided design of electronic systems without getting lost in the process. His constant encouragement, support and guidance were key to bringing this project to a fruitful completion. I am grateful to him for the training to carry out productive and directed research and for his friendship.

I would like to thank my associate advisor, Professor Michael J. Flynn for devoting precious time to monitor the progress of this research, and in reading this dissertation. Professor Krishna Saraswat has been very generous with his time to serve as chairperson for both my oral defense and the reading committees. I have also benefited from discussions with Professor Kunle Olukotun, who served on the defense and reading committees.

My sincerest thanks to my mentor Dr. Martin Freeman of Philips Research, Palo Alto for his constant guidance and a diligent reading of this dissertation. Needless to say, that any remaining mistakes are my sole responsibility. I would like to thank Uzi Bar-Gadda and Joe Kostelec of Philips Research, and Prof. Rick Reis and Carmen Miraflor of the Center for Integrated Systems for the honor to have been supported by the Philips Fellowship for the years 1992 and 1993.

This research builds upon the prior work of many people. I would like to thank all the other people involved in the synthesis project at Stanford. Of particular mention are David Ku, Frederic Mailhot, and Thomas Truong for writing the Olympus Synthesis System upon which the project is based. Claudionor Coelho wrote the simulator and contributed in numerous ways to this research work.

Many thanks are due to past and present members of our research group at Stanford. I am thankful to my colleagues David Ku, Frederic Mailhot, Maurizio Damiani, Polly

Siegel, Thomas Truong, Jerry Yang, David Filo and Claudionor Coelho for providing a supportive and productive environment during the course of my stay at Stanford. I would also like to take this opportunity to thank my friends outside our research group, Rohit Chandra and Kourosh Gharachorloo for many discussions and for providing the valuable 'non-CAD' feedback to this research. Many thanks are due to Ms. Lilian Betters for her valuable administrative support during the course of my stay at Stanford.

Many thanks to my colleague and one of the most critical examiners of this research, Dr. Mani Bhushan Srivastava of AT&T Bell Laboratories. I am greatly indebted to him for his ever sharp examination of the concepts and ideas presented in this dissertation. He has provided valuable and constructive feedback at every stage of this project that has enhanced the overall quality of the research work.

I would also like to thank Prof. Gaetano Borriello of University of Washington, Prof. Wayne Wolf of Princeton University and Prof. Daniel Gajski of University of California, Irvine for taking active interest in this research and providing a fertile environment to foster the growth of new ideas.

The time during which the ideas in this thesis were developed was an intense time for my family. It has all been made possible by the patience, love and support of my wife, Anne Usha, and the high spirits maintained by our son Anand. My regards to Atta and Mama for their family support and for giving me the most valuable companion in my life, my wife. Finally, my regards to my parents for their love and understanding, and to my brother Sanjay and sister Neena. They have all contributed in many ways to the person that I am today.

Financial support for this research was provided by a Fellowship provided by Philips and Center for Integrated Systems, and by NSF-ARPA under grant MIP 9115432.

Contents

Abstract **v**

Dedication **vii**

Acknowledgements **viii**

1 Introduction **1**

1.1 Design of Embedded Systems 4

1.2 Synthesis Solutions 5

1.3 Co-design and Co-synthesis 7

1.4 Motivations for Hardware-Software Co-synthesis 9

1.5 Applications of Hardware-Software Co-synthesis 13

1.6 The Opportunity of Co-synthesis 14

1.7 Architectures with Hardware-Software Components 16

1.7.1 Target system architecture 17

1.8 Scope and Contributions of Thesis 21

1.9 Outline of the Dissertation 22

2 Related Work **24**

2.1 CAD Systems for Hardware-Software Co-design 25

2.1.1 Ptolemy 26

2.1.2 CODES 28

2.1.3 Rapid prototyping using SIERRA 28

2.2 CAD for Hardware-Software Co-synthesis 29

2.2.1 COSYMA . . . 29
2.2.2 Use of non-deterministic finite state machines for co-design . . . 30
2.2.3 Co-synthesis from UNITY . . . 33
2.2.4 Interface co-synthesis . . . 33

3 System Modeling **35**
3.1 System Specification using Procedural HDL . . . 36
3.2 System Model and its Representation . . . 41
3.3 The Flow Graph Model . . . 43
3.3.1 Representation and definitions . . . 43
3.3.2 Hierarchy . . . 47
3.3.3 Execution semantics . . . 48
3.3.4 Implementation attributes . . . 50
3.4 Interaction Between System and its Environment . . . 59
3.4.1 Ports and communication . . . 59
3.4.2 Non-determinism in flow graph models . . . 60
3.5 $\mathcal{ND}$, Execution Rate and Communication . . . 61
3.6 Constraints . . . 65
3.6.1 Min/max delay constraints . . . 66
3.6.2 Execution rate constraints . . . 67
3.6.3 Specification of timing constraints . . . 69
3.7 Summary . . . 69

4 Constraint Analysis **72**
4.1 Scheduling of Operations . . . 73
4.2 Deterministic Analysis of Min/max Delay Constraints . . . 79
4.3 Deterministic Analysis of Execution Rate Constraints . . . 81
4.3.1 Procedure . . . 96
4.4 Min/max Constraints Across Graph Models . . . 99
4.5 $\mathcal{ND}$ Cycles in Constraint Graph . . . 102
4.5.1 Meaning of an $\mathcal{ND}$ cycle . . . 102
4.5.2 Problem formulation . . . 105

4.5.3 Use of buffers to extend bounds on loop index . . . 107
4.6 Probabilistic Analysis of Min/max and Rate Constraints . . . 109
4.6.1 Meaning of constraint satisfiability . . . 110
4.6.2 Index distribution and bounds on buffer depth . . . 113
4.7 Flow Graph as a Stochastic Process . . . 116
4.8 Summary . . . 124

5 Software and Runtime Environment 126
5.1 Processor Cost Model . . . 127
5.2 A Model for Software and Runtime System . . . 132
5.3 Estimation of Software Performance . . . 135
5.3.1 Operation delay in a software implementation . . . 136
5.4 Estimation of Software Size . . . 139
5.4.1 Operation linearization . . . 141
5.4.2 Estimation of register, memory operations . . . 147
5.4.3 Compiler effects . . . 154
5.4.4 Software data size and performance tradeoffs . . . 155
5.5 Software Synthesis . . . 155
5.6 Step I: Generation of Program Threads . . . 157
5.6.1 Implementation of inter-thread buffers . . . 164
5.7 Step II: Generation of Program Routines . . . 165
5.7.1 Concurrency in software through Interleaving: Coroutines . . . 166
5.7.2 Software implementation using description by cases . . . 166
5.8 Step III: Code Synthesis . . . 168
5.9 Issue in Code Synthesis from Program Routines . . . 168
5.9.1 Memory allocation . . . 169
5.9.2 Data types . . . 169
5.9.3 The C Standard Library . . . 170
5.9.4 Linking and loading compiled C-programs . . . 170
5.9.5 Interface to assembly routines . . . 171
5.10 Summary . . . 172

6 System Partitioning **174**
6.1 Partition Cost Model . . . 176
6.2 Local versus Global Properties . . . 181
6.3 Partitioning Feasibility . . . 183
6.3.1 Effect of runtime scheduler . . . 184
6.4 Partitioning Based on Separation of Control and Execute Procedures . . 188
6.5 Partitioning Based on Division of $\mathcal{NCD}$ Operations . . . 189
6.6 Partition Related Transformations . . . 193
6.7 Summary . . . 194

7 System Implementation **196**
7.1 Vulcan System Implementation . . . 196
7.1.1 Data organization in Vulcan . . . 199
7.1.2 Command organization in Vulcan . . . 200
7.2 Implementation of Target Architecture in Vulcan . . . 203
7.2.1 System synchronization . . . 203
7.2.2 Communication protocols . . . 207
7.2.3 Hardware-software interface architecture . . . 209
7.3 Co-simulation Environment. . . . 211
7.4 Summary . . . 217

8 Examples and Results **218**
8.1 Graphics Controller . . . 219
8.1.1 Implementation . . . 219
8.2 Network Controller . . . 225
8.2.1 Host CPU-controller interface . . . 225
8.2.2 Controller operation . . . 225
8.2.3 Controller architecture . . . 227
8.2.4 Network controller implementation results . . . 228

9 Summary, Conclusions and Future Work **232**
9.1 Future Work . . . 234

Bibliography **237**

A A Note on HardwareC **249**

B Bilogic Graphs **251**

C Processor Characterization in Vulcan **254**

D Runtime Scheduler Routines **256**

E Index of Notations **259**

List of Tables

1 *Operation vertices in a flow graph* 45
2 *Link vertices in a hierarchical flow graph* 47
3 *Basic instruction set* 129
4 *Addressing modes* 131
5 *Variable types and storage* 132
6 *Comparison of program thread implementation schemes* 167
7 *Vulcan (Rev 0) subsystems and commands.* 204
8 *A comparison of control FIFO implementation schemes* 223
9 *Graphics controller implementations.* 224
10 *Network controller instruction set* 227
11 *Network controller synthesis results using LSI library gates* 230
12 *Network controller synthesis results using Actel gates* 230
13 *Network controller software component* 230

List of Figures

1 *A design-oriented approach to system implementation.* 5
2 *A synthesis-oriented approach to system implementation.* 7
3 *Proposed approach to system implementation.* 8
4 *Synthesis approach to embedded systems.* 9
5 *Example of a mixed system implementation* 10
6 *DES Procedure* 11
7 *Bit permutations in DES Key Encryption* 12
8 *System Classification Based on HW/SW Components* 17
9 *Target System Architecture* 18
10 *Single chip realization of the target architecture* 20
11 *System Synthesis Procedure* 23
12 *Objects in* PTOLEMY 27
13 *Codesign flow in* COSYMA 30
14 *Co-design from Finite State Machines* 32
15 *Organization of Chapter 3* 37
16 *Linear code versus data-flow graph representations* 38
17 *Flow graph of process* `example`. 46
18 *Flow graph model for an error correction system.* 48
19 *Simulation of the graph model in Example 3.3.6* 57
20 *Shared memory versus message passing communication* 59
21 *Graph model properties* 63
22 *Communication across models.* 63
23 *Shared-memory versus message-passing implementations of loop operation.* 64

24 *General flow of constraint analysis.* 78
25 *Constraint Graph Model* 80
26 *Operation invocation interval.* 82
27 *Consecutive executions of an operation corresponds to traversal of a path in G.* 84
28 *Upward propagation of minimum execution rate.* 89
29 *Relationships between flow graphs* 99
30 *Graph model hierarchy* 101
31 *An $\mathcal{ND}$ cycle in the constraint graph* 102
32 *Types of loop operations.* 104
33 *Modeling an $\mathcal{ND}$ loop as a producer-consumer system* 105
34 *Buffer depth for exponential distributions ($\epsilon = 0.01\%$)* 116
35 *States of stochastic flow graphs.* 117
36 *Loops in a serialized model.* 120
37 *Loops in a fork.* 121
38 *Software model to avoid creation of $\mathcal{ND}$ cycles.* 134
39 *Software delay estimation flow.* 140
40 *Steps in generation of the software component* 156
41 *Use of enabling condition to build inter-thread dependencies.* 159
42 *Convexity serializations and possible thread implementations.* 161
43 *Generating fixed addresses from C-programs* 171
44 *Components of the partition cost model* 177
45 *Use of timing properties in partition cost function* 182
46 *Partitioning into Hardware Control and Software Execute Processes* . . . 189
47 *Partition of link vertices* 194
48 *Co-synthesis flow.* 197
49 *Data Organization in* VULCAN 199
50 *Vulcan subsystems and the Olympus Synthesis System* 201
51 *Flow of software synthesis in Vulcan* 202
52 *Control FIFO schematic* 205
53 *FIFO control state transition diagram* 206

54 *Hardware and Software Interface Architecture* 210
55 *Hardware and Software Interface Model* 210
56 *Event-driven Simulation of a Mixed System Implementation* 213
57 *Producer consumer system.* 214
58 *Example simulation: software producer, hardware consumer* 216
59 *Example simulation: software consumer, hardware producer* 216
60 *Graphics controller block diagram* 219
61 *Graphics controller implementation* 220
62 *Graphics controller software component using hardware control FIFO* . . 221
63 *Graphics controller software component using software control FIFO* . . 222
64 *Graphics controller simulation* 223
65 *Network controller block diagram* 226
66 *Format of an ethernet frame* 228
67 *Network controller implementation* 229
68 *Network controller simulation* 231

Chapter 1

Introduction

Recent years have seen remarkable growth in the design and use of digital systems in several application areas. Digital systems are designed for two major *classes* of applications: *general-purpose* and *application-specific* systems. General-purpose systems are not designed for any specific applications but can be *programmed* to run different applications. The most common use of general-purpose systems is in computing applications. Examples of these systems are computers such as workstations.

In contrast, *application-specific* systems are designed for dedicated applications. Examples of such systems can be found in medical instrumentation, process control, automated vehicles control, and networking and communication systems. As these systems are contained within a larger non-electronic environment, these are commonly referred to as **embedded systems**. Embedded computer systems have been applied to tasks erstwhile handled by electronic or electro-mechanical *non-computing* systems. As a result, the volume of embedded electronics market has grown. For the year 1991, the industrial and medical electronics market alone accounted for $31 billion compared to the general purpose computing systems market of $46.5 billion [JJ93].

This growth has been fueled by the advent of microprocessors, the primary compute element in a system. Microcontrollers, a derivative of microprocessors, are now beginning to be used in many embedded systems. For the year 1991, the market for microcontrollers amounted to $4.6 billion and has been rising at a 18% annual growth rate compared to a 10% annual growth rate of general-purpose systems [JJ93].

While there has been notable growth in the use and application of embedded systems, improvements in the design process for such systems have not kept pace, leading to a gap in the evolution of component technology and its application in embedded computing systems. While new processors and programmable/reprogrammable integrated circuits are announced every six months with an average performance boost of 50% per year, it may be several years before such components find use in embedded computer systems. This is in contrast to the design of general-purpose computing systems that have largely kept pace with advances in component technology related to processors, memory or integrated circuits. Currently, approximately 80% of the microcontrollers used in embedded systems are 4- and 8-bit processors of old generations [Bad93]. Of the total $4.6 billion microcontroller market for 1991, 32-bit processors account for less than 4% or $184 million, despite the fact that such processors have been commonplace since 1985 and almost all advances in processor technology since then have been concentrated in the design of 32-bit processors.

Examining the cost analysis for semiconductor manufacturing, including package and testing, the total chip cost for a die size of 1x1 sq. cm. comes to an average of $27 versus $7 for a 0.25x0.25 sq. cm. die [HP90]. Thus, chip manufacturing cost is not always the dominant factor in overall chip pricing. Historically, it has been observed that the prices of single-chip processors stabilize to a certain level in the $10-50 range regardless of the introduction price of the processor. This drop in price is more strongly related to the advancements on the technology learning and yield curves than to market dynamics. Typically this price stabilization occurs within two years of the introduction of a processor.

There are several reasons for this discrepancy in the advancement of embedded versus general-purpose systems. More often, the embedded system is not the most visible part of the application and, therefore, its implementation inefficiencies are often overlooked. Component prices and manufacturing/maintenance cost of such systems is often cited as reasons for relatively slow proliferation of advanced components in such systems. However, as explained, the cost/price stabilization for processors occurs much sooner than their proliferation in the embedded systems. Further, instead of using multiple 4-, 8- or 16-bit processors, the trend in embedded system design is to use 32-bit processors

despite increased system costs. The use of 32-bit controllers in embedded applications, though a tiny 4% of the total volume, is increasing at an annual rate of 52% as against the overall growth of 18% for the overall embedded controller market [JJ93].

In summary, even though there is a greatly recognized need for the use of advanced 32-bit processors in embedded applications, their proliferation in terms of total volume has been lagging behind the growth in the 32-bit processor market. Further, the price stabilization for these advanced processors occurs much earlier than their bulk use in embedded systems. Thus, the proliferation of advanced 32-bit processors in embedded systems does not appear to be limited by component cost considerations alone.

The chief reason for the slow proliferation of advanced components into embedded systems is **the long design time and high cost of design** of such systems. Since embedded systems are tailored to specific applications, the design cost per unit volume is higher for embedded systems. Therefore, such systems stand to gain most from advances in the design process that shortens the design time and improves performances by leveraging the use of newer and advanced components. Based on this hypothesis, this thesis examines the problems in system design and provides solutions to speed up the design process by developing *synthesis* techniques. The difference in design and synthesis techniques is discussed a little later in this chapter. But first we briefly examine some of the commonly used terms associated with the design of computer systems.

An electronic system consists of a set of *interacting* components. A digital electronic system implements its primary functions using components that react to and produce discrete objects. A component of a system may be a system in itself. At the lowest-level (leaf-level) components tend to be more functionally homogeneous than the systems that use these components. For example, a digital computer system consists of software, processor, memory and peripheral input/output components. For this reason, systems are often called *heterogenous systems*.

A component functionality is classified as either a *computation* or *communication*. As the terms suggest, a communication functionality relates to operations involving input and output operations, with the rest being computations. Both computation and communication functionalities can be implemented in a *synchronous* or an *asynchronous* manner. Synchronous communication refers to a *constant phase-relationship* between two or more

input and/or output operations. Asynchronous communication, on the other hand, refers to input/output operations that have no or variable phase-relationships. A synchronous implementation uses a *global* mechanism such as a clock, whereas asynchronous computations are characterized by the absence of such *global synchronization* mechanisms.

This thesis is targeted at exploring the implementation of embedded digital systems with synchronous computation and communication components that are targeted for specific applications. In addition to being application-specific, such systems are also designed to respect constraints related to the relative timing of their actions, hence these systems are referred to as *real-time embedded systems*. The application-specific nature of these systems often requires custom hardware circuits and programs to run on a general-purpose processor hardware. This personalization is commonly referred to as programmability in **hardware** and **software** respectively. Since the components may be re-used, we are interested in hardware and software components that can be *reprogrammed* to suit applications or changes and upgrades in an application.

1.1 Design of Embedded Systems

While there have been tremendous advancements in the design of the general purpose components of an embedded system, the design of the hardware and software components to achieve its programmability has not changed much over the years. Software programmability is achieved by manually writing software often in processor assembly language. Similarly, hardware programmability is achieved by manual design using gate-level circuits or low to medium-scale integration circuits.

There are several challenges in the design and the analysis of time-constrained embedded systems that prolong the design process. Important among these are the problems of performance estimation, selection of appropriate components and verification of such systems for functional and temporal properties. In practice, such systems are created using a **design-oriented** approach. The system is specified by a collection of its functionalities which are then implemented by a choice of appropriate components.

For instance, consider the design of a network controller shown in Figure 1. The controller is connected to a serial line and a memory. The purpose of the controller is to

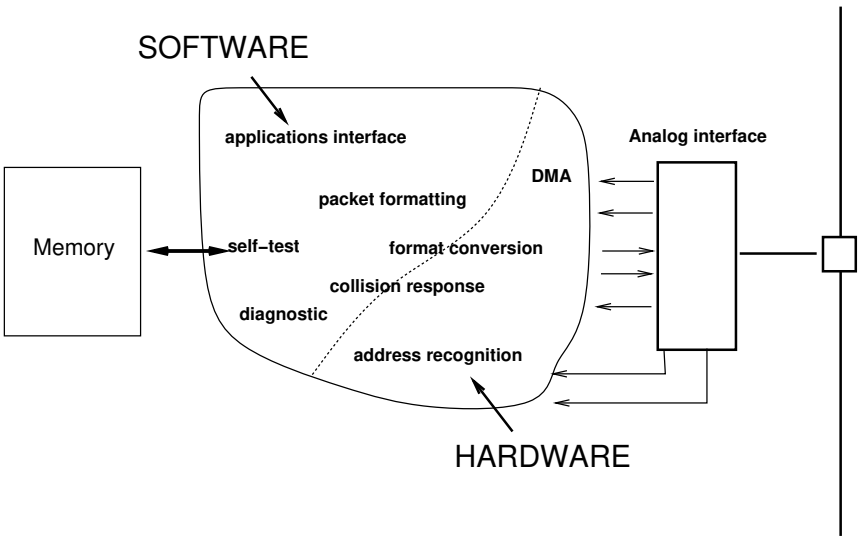


Figure 1: *A design-oriented approach to system implementation.*

receive and send data over the serial line using a specific communication protocol (such as CS/CD protocol for ethernet links). The decision to map functionalities into dedicated hardware or implement them as programs on a processor is usually based on estimates of achievable performance and implementation cost of the respective parts.

There are several limitations to this approach. The division of functionality between components is based on the designer's experience and takes place early on in the design process. This often leads to portions of the design that are either under- or over-designed with respect to their required performance. More importantly, due to the ad-hoc nature of the overall design process, there is no guarantee that a given implementation meets the required system performance (except possibly by overdesigning).

1.2 Synthesis Solutions

In contrast to design-oriented solutions, a methodical approach to system implementation can be formulated as a **synthesis-oriented** solution which has been enormously successful in the design of individual integrated circuit chips (*chip-level* synthesis). Instead of using a specification as a set of loosely-defined functionalities, a synthesis approach for hardware proceeds with systems described at the *behavioral level* by means of an

appropriate specification language. While the choice of finding a suitable specification language for digital systems is a subject of on-going research, the use of procedural *hardware description languages* (HDLs) to describe integrated circuits has been gaining wide acceptance in recent years.

A synthesis-oriented approach to digital circuit design takes a behavioral description of circuit functionality and attempts to generate a gate-level implementation that can be characterized as a purely hardware implementation (Figure 2). Recent strides in high-level synthesis have made it possible to synthesize digital circuits from high-level specifications and several such systems are available from industry and academia [BCM+88, CR89, RMV+88, CPTR89, TLW+90, MKMT90, WTH+92]. The outcome of synthesis is a gate-level or geometric-level description that is implemented as a single chip or multiple chips.

An alternative to hardware synthesis of system prototypes would be to build a *software prototype* of the system functionality. Such implementations lie on the opposite end of the cost-performance spectrum (Figure 2). Here cost refers to system development cost and performance is the time-related performance of the prototype. A software prototype consists of a completely software specification based on a programming language that is sometimes enhanced to support the structural interconnection of language objects [BL90a, BL90b]. An example of a software prototyping system is the RAPIDE prototyping system [LVBA93]. Software prototypes are rather quick to build and are often used for verifying system functionality. Because the prototypes are primarily targeted for simulations, there is a limit to the resolution of time-scale of events that can be used. Therefore, the timing performance of software prototypes often falls short of what is desired for time-constrained system designs.

In summary, there are several limitations of existing synthesis-oriented solutions to system implementation. For synthesized hardware solutions, as the number of gates (or logic cells) increases, such implementations require the use of semi-custom or custom design technologies with their associated increases in cost and design turn-around time. Therefore, for large system designs, synthesized hardware solutions tend to be fairly expensive depending upon the choice of technology required for chip implementation. As mentioned earlier, software solutions on the other hand, often fail to meet constraints

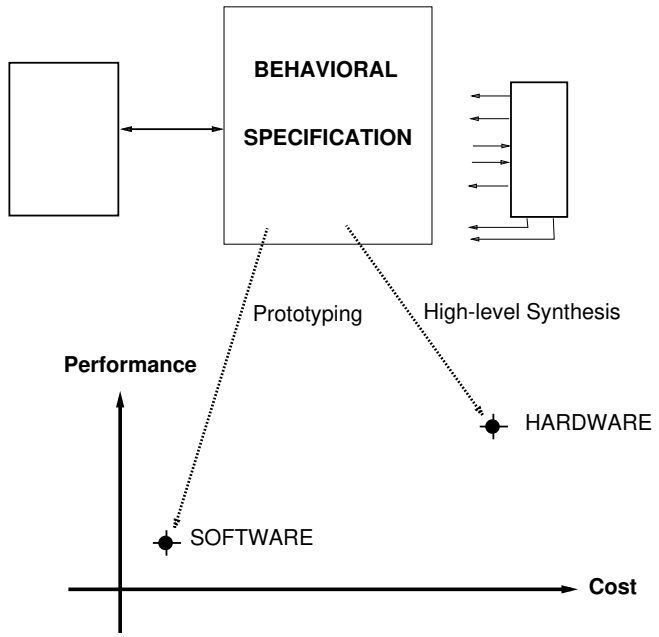


Figure 2: *A synthesis-oriented approach to system implementation.*

on timing performance.

1.3 Co-design and Co-synthesis

Synthesis-oriented approaches to system implementation provide systematic and rapid evaluation of implementation alternatives. System cost and performance tradeoffs dictate a choice between synthesized hardware solutions or software prototypes. But we do know from our practical experience that cost-effective designs use a mixture of hardware and software to accomplish their overall goals (Figure 1).

This provides sufficient motivation for attempting a system implementation that contains both hardware and software components. This is commonly referred to as the process of **hardware-software co-design** where both components are designed together. The input specification in co-design may consist of a single or a collection of *heterogenous* specifications.

A further development in this direction would be a **co-synthesis** approach that attempts to provide mixed hardware-software implementations using synthesis techniques. Such

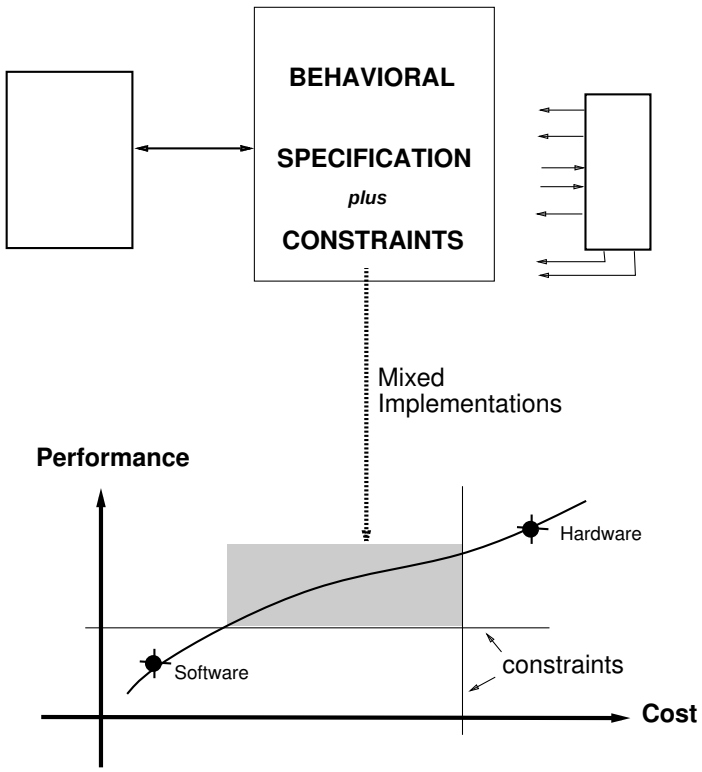


Figure 3: *Proposed approach to system implementation.*

an approach would benefit from a systematic analysis of design trade-offs that is common in synthesis, while at the same time creating systems that are cost-effective. One way to accomplish this task would be to specify constraints on the cost and the performance of the resulting implementation (Figure 3).

This thesis presents a synthesis approach to systematic exploration of system designs that is driven by the constraints. This work is built upon high-level synthesis techniques for digital hardware [MKMT90] by extending the concept of a resource needed for implementation. Figure 4 shows the essential aspects of this approach. A behavioral specification is captured into a system model that is partitioned for implementation into hardware and software. The partitioned model is then synthesized into interacting hardware and software components for the target architecture shown in Figure 9. The target architecture uses one processor and application-specific hardware. The target architecture is described in further detail in Section 1.7.1.

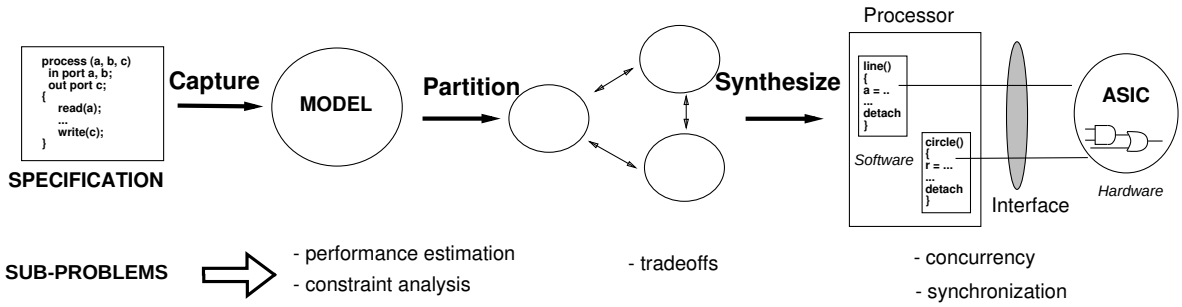


Figure 4: *Synthesis approach to embedded systems.*

1.4 Motivations for Hardware-Software Co-synthesis

Most digital functions can be implemented by software programs. The major reason for building dedicated application-specific hardware (ASICs) is the satisfaction of performance constraints. These performance constraints can be on the overall time (latency) to perform a given task, or more specifically on the timing to perform a subtask and/or on the ability to sustain specified input/output data rates over multiple executions of the system model. The hardware performance depends on the results of operation scheduling and on the performance characteristics of individual hardware resources. The software performance, defined as the number of cycles that it takes the processor to execute a routine, depends on the number of instructions the processor must execute and the cycles-per-instruction (CPI) metric of the processor. In general, application-specific hardware implementations tend to be faster since the underlying hardware is optimized for the specific set of tasks. However, in the absence of stringent performance constraints, for a given behavioral description of an ASIC machine, some parts (subroutines) of it may be well suited to a commonly available re-programmable processor (e.g., 6502, 68HC11, 8051, 8096 etc) while others may take too long to execute. For instance, most general purpose CPUs deal with byte-size operands whereas many ASIC controllers contain bit-oriented operations resulting in unnecessary overheads when the operations are implemented entirely in software. However, the software implementations do provide the ease and flexibility of reprogramming for the possible price of loss of performance.

Example 1.4.1. Data encryption controller.

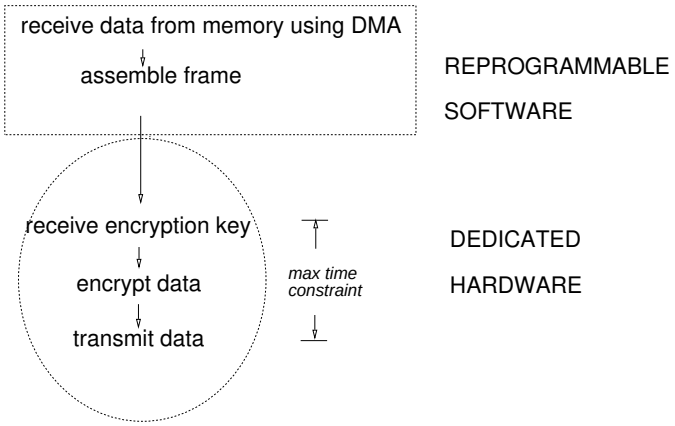


Figure 5: *Example of a mixed system implementation*

To be specific, consider the design of a data encryption/protocol controller chip, based on the DES (Data Encryption Standard) protocol used by commercial banks or the AES (Audio Engineering Society) protocol used for communication between digital audio devices and computers. In Figure 5, the DES transmitter takes data from memory using a DMA controller, assembles the frame for transmission, encrypts the data after it receives the key and transmits the encrypted data. The encryption protocol requires that the encrypted data be transmitted within a certain time duration of receiving the encryption key.

In the DES protocol, a 64-bit encryption key is used to transform 64 bits of 'plaintext' into 64 bits of encrypted text. Here we present only the relevant aspects of the encryption process. For details on the standard and algorithms the reader is referred to [oS88] [SB88]. The encryption key contains 8 parity bits which are removed before the encryption process thus deriving a 56-bit encryption key. As shown in Figure 6, the entire encryption process consists of 18 permutation stages including an initial and a final stage which do not require any key. The 16 intermediate steps are key-controlled. The first and last stages are simple permutations. The 16 48-bit keys required for intermediate stages are derived from the original 56-bit key. Thus there are two separate 16-stage operations: a) generation of the 48-bit encryption key, and b) use of encryption key to manipulate 64-bit data. Here we consider the first operation, that is the generation of the encryption key, though a similar argument can also be made for the data manipulation operations which consists of rotation, permutations, `xor` and table lookup.

The encryption key algorithm transforms the 56-bit input key buffer (known as shifted key buffer, SKB) into a 48-bit key which is organized as an 8-byte key buffer (KB) such that only 6 bits from each byte of the KB are used in the key.

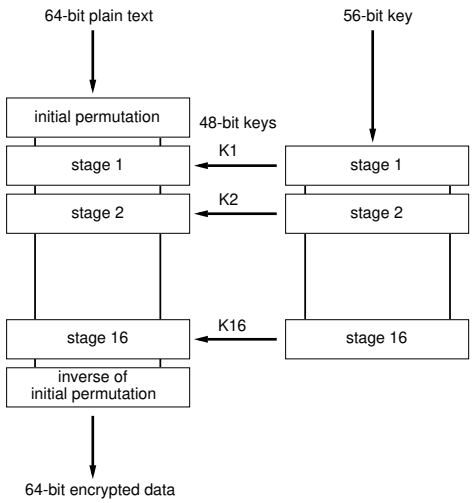


Figure 6: *DES Procedure*

Thus each stage of the 16-stage key generation algorithm consists of 48 permutation operations on the shifted key buffer as illustrated by the algorithm below:

```
clear 64-bit key buffer
for i = 1 .. 48 do {
  isolate bit i of the shifted keyed buffer
if (bit == 1)
    set key buffer bit pc2(i) using permuted choice table, pc2
}
```

Software implementations of this encryption key algorithm vary from 300 to 3000 instructions depending on the level of bit-oriented operations supported. This is in sharp contrast to the hardware implementation in which each stage can be accomplished in a single cycle by building the permutation into an interconnection network as shown in Figure 7. Therefore, a hardware implementation of the algorithm would require 16 cycles. □

Thus, hardware and software implementations vary widely in speed. For designs that are dominated by bit-oriented operations, dedicated hardware implementations are preferred, whereas it may take too long to execute these operations as a sequence of instructions on most processors, thus violating the constraints on timing performance of the controller. Whereas implementing the complete protocol controller on dedicated hardware may be too expensive, an implementation which uses a re-programmable component may satisfy performance requirements and at the same time provide the ease and flexibility of programming in software.

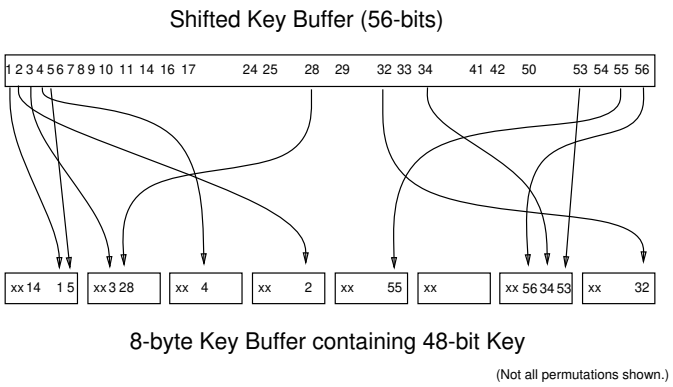


Figure 7: *Bit permutations in DES Key Encryption*

While bit-wise shifting and `xor` operations lead to slower software implementations, the implementation of byte-oriented data-intensive operations with the use of structured memory is considerably faster. Such implementations are often competitive with corresponding hardware implementations. Consider the following example.

Example 1.4.2. Cyclic redundancy code computation.

Consider a 16-bit CRC-CCITT computation using the polynomial $x^{16} + x^{12} + x^5 + 1$. With the addition of every byte of data, the new CRC is clearly a function of 8-bits of the old CRC and the new byte of data. This function is precomputed and stored in a 256-entry table. A byte-wise implementation using two 256-byte tables, as described by the following pseudo-code, when coded in assembly can achieve the 16-bit CRC computation in 7 instructions per byte.

```
typedef byte char;

byte Table_low[256], Table_high[256];
byte Temp, data, CRC_low, CRC_high;

Temp = data xor CRC_low;
CRC_low = Table_low[Temp] xor CRC_high;
CRC_high = Table_high[Temp];
```

The actual latency of computation is strongly dependent on the instruction-set architecture (ISA) of the target processor. The best implementation of the above pseudo-code on an Intel 8086 processor computes 16-bit CRCs in 9 instructions, a Motorola 68K implementation in 11 instructions and a RISC-based implementation in 14 instructions.

In contrast to a table-driven software implementation, a hardware implementation of the CRC typically consists of a 16-bit shift register with `xor` taps at locations

dictated by the polynomial (i.e., positions 0, 5 and 12). Then the incoming bit stream is shifted from the right into the shift register by left shifting the register. Any time a 1 bit gets shifted off the left end of this register, the register contents are replaced by an `xor` with the polynomial (equivalent to a modulo-2 subtraction operation). This implementation results in a CRC computation rate of 8 cycles/byte. □

1.5 Applications of Hardware-Software Co-synthesis

The hardware-software co-synthesis techniques formulated in this thesis can be used for following applications.

1. **Design of cost-effective systems:** The overall *cost* of a system implementation can be reduced by the ability to use already available general purpose re-programmable components while reducing the number of application-specific components.

2. **Rapid prototyping of complex system designs:** A complete hardware prototype of a complex system is often too big to be implemented using field programmable gate-array (FPGA) technologies. For such systems a mask programmable or even a custom hardware realization is required. With the identification of the time critical hardware section, the total amount of hardware to be synthesized may be reduced significantly, thus making it feasible for rapid prototyping. A feasible partition that shifts the non performance-critical tasks to software programs can be used to quickly evaluate the design.

3. **Speedup of hardware emulation software:** During their development phase, many system designs are often modeled and emulated in software for test and debugging purposes. Such an emulation can be assisted by dedicated hardware components which provide a speedup on the emulation time.

Rapid prototyping and hardware emulation are two opposite ends of the system synthesis objective. Rapid prototyping attempts to minimize the application-specific component to reduce design time, whereas hardware emulation attempts to maximize the application-specific component to realize maximum speed-up.

1.6 The Opportunity of Co-synthesis

This thesis explores the opportunity of achieving hardware-software co-synthesis by formulating it as a problem of system partitioning into application-specific and re-programmable components. We can also view it as an extension of high-level synthesis techniques to systems with generic re-programmable resources. Nevertheless, the overall problem is much more complex and it involves, among others, solving the following sub-problems:

1. Modeling the system functionality and performance constraints.

   System modeling refers to the *specification* problem of capturing important aspects of system functionality and constraints to facilitate design implementation and evaluation. Most hardware description languages attempt to describe a system functionality as a set of computations performed by a computing element and as interactions among computing elements. Among the important issues relevant to mixed system designs are:

   - explicit or implicit concurrency in specification
   - model of communication - shared memory versus message-passing
   - control flow specification or *scheduling* information

   There is a relationship between concurrency in specification and the *natural* partitions in the system descriptions. Typically, languages that contain explicit partitioning via control flow breaks, find it difficult to specify concurrency explicitly. Concurrency information is then obtained by performing a dependency analysis whose complexity depends on the model of communication used. We consider the relevant modeling issues in Chapter 3.

2. Choosing the granularity of the hardware-software partition.

   The system functionality can be handled either at the functional abstraction level where a certain set of *high-level* operations is partitioned or at the process communication level where a system model composed of interacting process models

is mapped onto either hardware or software. The former attempts fine grain partitioning while the latter attempts a *high-level library binding* through coarse-grain partitioning.

3. Determining the feasible partitions of application-specific and re-programmable components.

   The so-called problem of *hardware-software partitioning*. This delineation is influenced by issues such as analog interfaces that require a specialized hardware interface. However, for operations that can be implemented either in hardware or in software, the problem requires a careful analysis of the flow of data and control in the system model.

4. Specifying and synthesizing the hardware-software interface.

5. Implementing software routines to provide real-time response to concurrently executing hardware modules.

6. Using synchronization mechanisms for software routines and synchronization between hardware and software portions of the system.

One important issue that needs to be resolved before addressing the co-synthesis subproblems is choice of a target system architecture. By target system architecture we mean general organization of its components. As with specification, target architectures for embedded systems are not universally defined or accepted. This is in sharp contrast with single-chip systems where a single synchronous data-path/controller chip organization is almost always implied unless otherwise mentioned.

The choice of a system architecture is not a trivial task due to the great impact of system organization on system cost and performance. Further, a target architecture is strongly influenced by the *specific application* to which the system is targeted. This issue, though important, is peripheral to the co-synthesis problem that this thesis seeks to address. Therefore, we choose an architecture that preserves essential features of mixed systems while leaving the specific details as a co-design problem that must be solved in the context of an application. In order to put it in proper perspective, we present

our target system architecture in the context of the organization of some of the familiar computer systems.

1.7 Architectures with Hardware-Software Components

As mentioned earlier, most digital computer systems are either *general-purpose* or *embedded*. General-purpose computer systems contain some form of storage that can be altered (reprogrammed) by the user under software control. On the other hand, embedded systems are usually hard-wired for certain specific tasks such that the degree of 'reprogrammability' varies from none to the changing of parameters of some existing sequential control. An embedded system may have a dedicated controller (a sequencer) or a microcontroller programmed to sequence operations. Most of these systems contain storage (program or data) which is relatively small and cannot be easily altered. Microcontrollers are essentially general purpose microprocessors with on-board memory for program and data storage. The ability to reprogram a computer system is related to the versatility of its primitive operations, or the instruction-set of the microprocessor or microcomputer used in the system. In our terminology, we refer to a microprocessor or a microcontroller as a *reprogrammable component* or simply as a *processor*. The specific sequence of instructions needed for a particular application to be executed by the reprogrammable component is referred to as the software component.

Thus, in broad terms, a digital system can be thought of as consisting of two components: *software* as a program in an on-board RAM or ROM and *hardware* as the underlying interconnection of special-purpose blocks. Based on this distinction, Figure 8 shows compositions of some familiar systems. The hardware component in a system design may be manually designed or synthesized automatically using a silicon compiler. The software component of a system may consist of microcoded routines, or machine-level programs used in embedded control systems or high-level programs used in special-purpose machines. Note that some system designs, most notably single-chip microprocessors, use microprogramming simply as a design technique for implementation of hardware control. This is different from the software *necessary* to achieve system functionality, as in microprogramming of functional algorithms in the case of mainframe

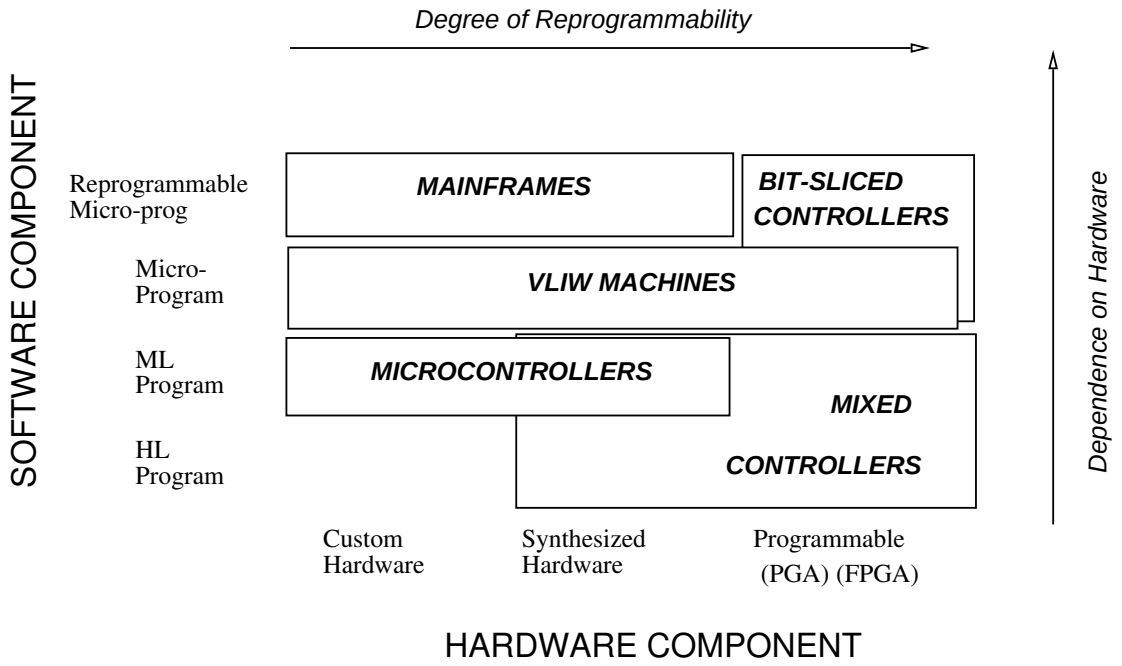


Figure 8: *System Classification Based on HW/SW Components*

machines. Conventionally, machine-level and high-level programs manipulate user data-structures, while microprograms manipulate hardware resources. In the case of the mixed controller designs proposed in this dissertation, we use machine-level programs to perform both activities. The co-synthesis approach proposed in this work addresses the design problem of mixed controllers shown in Figure 8.

1.7.1 Target system architecture

For the purposes of developing a co-synthesis approach, we choose the target architecture shown in Figure 9 that consists of a processor assisted by application-specific hardware components.

The application-specific hardware is not pipelined, for the sake of simplifying the synthesis and performance estimation task for the hardware component. Even with its relative simplicity, the target architecture is applicable to a wide class of applications in embedded systems used in medical instrumentation, process control, vehicular control,

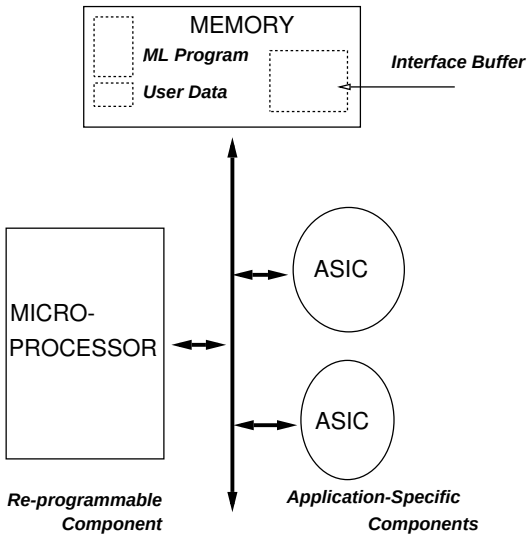


Figure 9: *Target System Architecture*

and communications. The following lists the assumptions relating to the target architecture. These assumptions are made in order to keep the relevant synthesis issues subject to a systematic approach, while at the same time retaining generality and effectiveness of the target architecture. Many of these assumptions can be dropped in a larger system co-design methodology without affecting the underlying co-synthesis approach developed here.

- We restrict ourselves to use of a *single re-programmable component*. We make this simplifying assumption in order to make the synthesis tasks manageable. The presence of multiple re-programmable components requires additional software synchronization and memory protection considerations to facilitate safe multiprocessing. Multiprocessor implementations also increase the system cost due to requirements for additional system bus bandwidth to facilitate inter-processor communications. These issues, though important, are not directly relevant to the focus on system co-synthesis problem addressed in this work.

- The memory used for program and data-storage may be on-board the processor. However, the interface buffer memory needs to be accessible to all of the hardware

modules directly. Because of the complexities associated with modeling hierarchical memory design, we consider only the case where all memory accesses are to a single level memory, i.e., outside the re-programmable component. The hardware modules are connected to the system address and data busses. Thus, all the communication between the processor and different hardware modules takes place over a shared medium.

- The re-programmable component is always the bus master. Almost all re-programmable components come with facilities for bus control. The inclusion of such functionality on the application-specific component would greatly increase the total hardware cost.

- All the communication between the re-programmable component and the application-specific circuits is done over named channels whose width (i.e. number of bits) is the same as the corresponding port widths used by read and write instructions in the software component. The physical communication takes place over a shared bus.

- The re-programmable component contains a 'sufficient' number of maskable interrupt input signals. For the purpose of simplicity, we assume that these interrupts are unvectored and there exists a predefined destination address associated with each interrupt signal.

- The application-specific components have a well-defined *RESET* state that is achieved through a system initialization sequence.

Figure 10 shows a single chip realization of the target architecture. The processor in this realization refers to the processor *core* of a general-purpose microprocessor. Physical implementation of the ASIC may be achieved using standard cells or gate array circuits. The interface between the processor and the ASIC refers to the hardware portion of the interface circuitry (for details on this see Section 7.2.3). The memory consists of program ROM plus any RAM buffers need for the interface. Finally, the system interface may be composed of analog I/O circuits such as A/D and D/A converters, direct-memory access

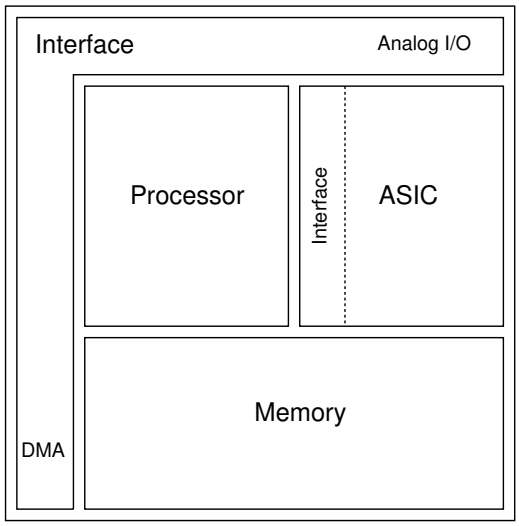


Figure 10: *Single chip realization of the target architecture*

(DMA) circuits, and any possible data width conversion circuits (serial-to-parallel and parallel-to-serial).

It is important to note that the final system implementation may or may not be a single-chip system design, depending on availability of the re-programmable component either as a macro-cell or as a separate chip. Further, the approach outlined in this report can also be used for alternative target architectures.

The key concept in any realization of the target architecture is the fact that a (general-purpose) processor is used as merely another **resource** *to realize system functionality. The emphasis is not to build a system around a processor, instead the emphasis is to use a processor to reduce the size of the ASIC circuitry.* At first glance, these two approaches may appear to lead to the same implementation. However, the difference is in the emphasis on the utilization of the processor to implement system functionality. For systems that are built around a given processor (or processors), the chief objective of system design is to exploit processor functionality and utilization to the fullest extent, as is the case in general-purpose computing systems. This often requires design decisions that are difficult, if not impossible, to capture in a synthesis-oriented solution. In contrast, when using the processor as another resource, the objective is to reduce the ASIC size while meeting constraints where actual utilization of the processor is of secondary importance.

Due to the emphasis on devising a synthesis-oriented solution to achieve embedded system design, the resulting implementations have some limitations in their scope and applicability. These limitations are due to assumptions made on the runtime system and system interfaces in order to reduce the complexity of the embedded system design details. The assumptions and limitations are described in their context in corresponding chapters.

1.8 Scope and Contributions of Thesis

The following lists the goals and contributions of this dissertation:

- Development of a model for capturing hardware and software properties. The model is based on a graph-based representation of operations and dependencies and on the relationship of computation rates to the associated communication mechanisms.
- Formulation of rate constraints for high-level synthesis purposes and analysis of feasible hardware-software partitions in the context of general timing constraints.
- Development of partitioning schemes that capture both spatial and temporal properties of the partitioned systems.
- Development of a runtime system software that is suitable for co-synthesis.
- Development of model transformations to meet rate constraints, in particular program transformations to improve latency and throughput.
- Design of a low-overhead hardware-software interface architecture.
- VULCAN - a CAD tool for exploring system-level designs.

Part of the subject matter addressed in this thesis has been presented in following publications [GM90, GM91, GM92, GCM92b, GCM92a, GM93, GCM94].

1.9 Outline of the Dissertation

This thesis is organized according to the problem taxonomy described in Section 1.6. Chapter 2 briefly presents an overview of related work in system design and computer-aided design techniques developed for system synthesis. The organization of the rest of thesis can be best explained by relating it to the organization of our co-synthesis CAD system, VULCAN shown in Figure 11. The input to our synthesis system is an algorithmic description of system functionality described in a hardware description language (HDL). The HDL description is compiled into a system graph model based on data-flow graphs. Chapter 3 describes the features and properties of our system model. Chapter 4 describes constraint modeling and analysis techniques to determine feasibility of hardware-software implementations.

In Chapter 5 we discuss issues and techniques in the generation of software and its associated runtime environment. We introduce the concept of a threaded software implementation which is shown to observe constraint satisfiability properties discussed in Chapter 4. In Chapter 6 we define the problem of system partitioning and present an approach to partitioning of systems for hardware-software co-synthesis. In Chapter 7 we discuss issues in system synchronization, how synchronization is achieved between heterogeneous components of a system design. Here we also present an overview of the VULCAN system. The resulting mixed system design consists of an assembly code for the software component, and a gate-level description of the hardware and hardware-software interface. This heterogenous description is simulated using the program POSEIDON that is described elsewhere [GCM92b].

Chapter 8 describes case studies in hardware-software co-synthesis and results. Chapter 9 presents conclusions and directions for future research.

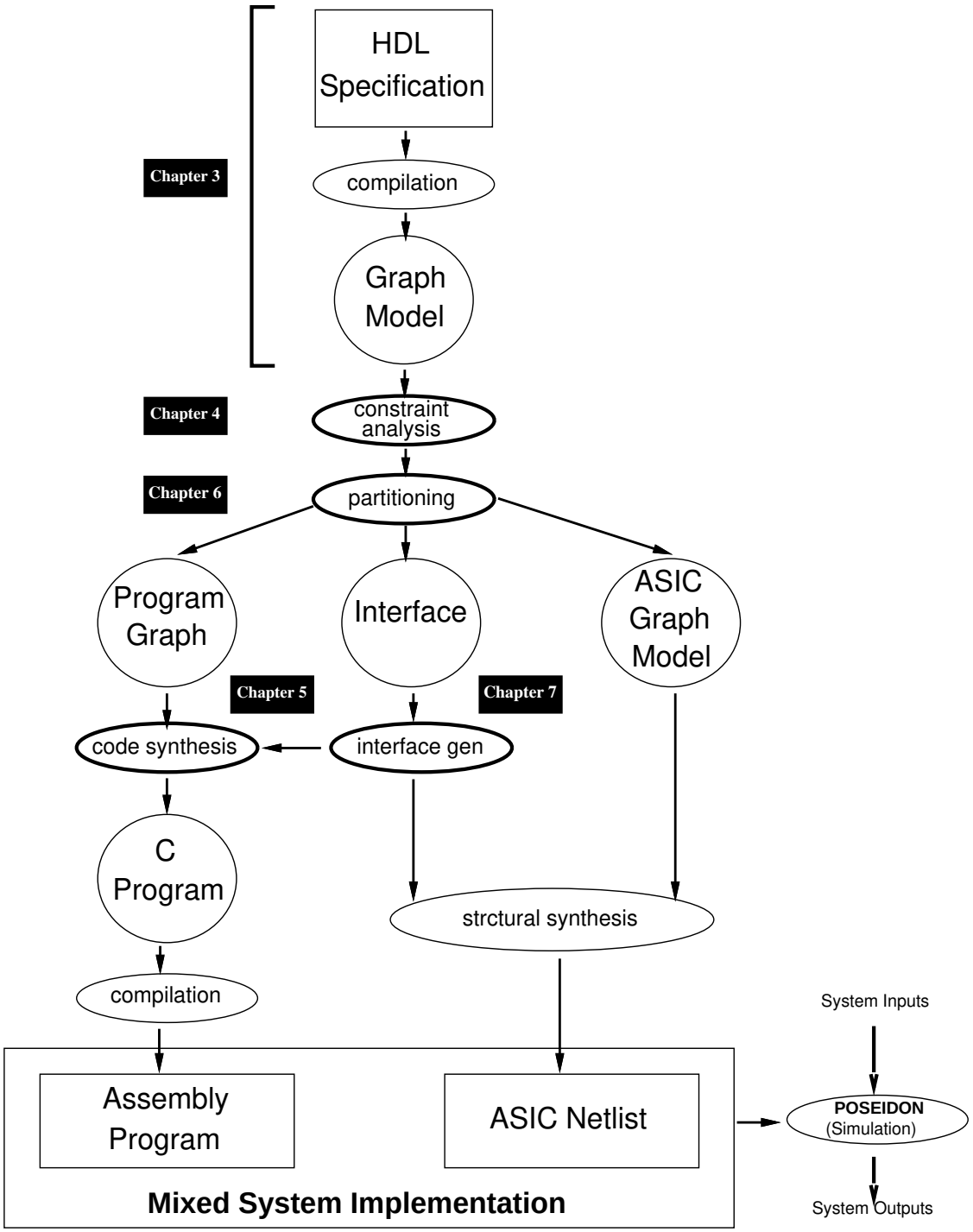


Figure 11: *System Synthesis Procedure*

Chapter 2

Related Work

This chapter reviews important developments in the area of system design and synthesis. The issue of co-design of hardware and software often appears in the larger context of system design. Computer architects often tradeoff the implementation of an instruction in hardware versus its implementation in software as a sequence of available instructions. This flavor of the co-design problem addresses the issue of design of software and hardware *upon* which the software runs. This is clearly different from the notion of hardware and software defined in the previous chapter, where the software runs on a predesigned hardware. The idea of hardware-software co-design has even been applied to the *process* of system design [BV92].

We briefly review some of the novel architectures that consist of a mix of hardware and software. Programmable active memories, PAM [BRV89], use a network of basic cells which are programmed for specific applications. The Map-oriented Machine (MoM) [HHW89] belongs to a class of system architectures for implementing systolic algorithms. MoM's relevance to system co-design is highlighted by its reliance on reprogrammable technologies to achieve performance speedups. Indeed, its derivative work on *xputer*[HHR+91] attempts to use MoM architectural principles in prototype implementation of non-systolic algorithms. MoM is characterized by data-driven execution streams. This key advantage is achieved by replacing register and ALU combinations in sequential processors by a logic unit, called the problem-oriented logic unit, that uses RAMs, PLDs and other programmable hardware. QuickTurn [Wal90] and PiE systems

use reprogrammable hardware to create system prototypes. The primary advantage of these systems is the short amount of time it takes to create and modify these prototypes that may not provide the intended system timing performance, but these prototypes are considerably faster than their equivalent software prototypes.

Another area where the co-design problem has been studied is in the design and analysis of 'real-time systems'. Real-time systems span a wide variety of applications and can be fairly complex. Performability analysis of real-time systems, defined as analysis of system performance metrics over finite time intervals, is one of the key analytical tools.

Work in the computer-aided approach to system design is relatively new. Recent interest in system synthesis has been stimulated by the relative success and maturity of chip-level synthesis tools, and emergence of synthesis approaches at levels of abstraction higher than logic-level and RTL-level circuit descriptions. CAD related work falls under two broad categories:

1. **Generic CAD for supporting hardware-software co-design.** These approaches generally recognize the difficulty in addressing all parts of the system design problem in a unified framework. Therefore, these systems concentrate on providing a **frame-work** to support the process of system design.

2. **Specific CAD for hardware-software co-synthesis.** Work in this area concentrates on providing CAD solutions to specific synthesis sub-problems. Most of these solutions are devised under specific restrictions on system implementation.

2.1 CAD Systems for Hardware-Software Co-design

A CAD system refers to an integrated collection of tools that conform to an overlaying methodology usage of these tools. The overall goal of a CAD system is to improve the process of system analysis and design. In all these systems, trade-offs are made among the following metrics:

1. **Analyzability** - the ability to analyze a system design for its functional and performance properties,

2. **Simulatability** - the ease in arriving at a complete system simulation,

3. **Implementability** - the ability to implement (design or synthesize) a system from its specification.

2.1.1 Ptolemy

PTOLEMY [BHLMar] [KL93] is a framework for the simulation, prototyping and software synthesis of digital signal processing systems. Due to its application focus on the DSP domain, the reprogrammable components in system design are chosen from a set of general-purpose DSP processors (or equivalent cores), such as Motorola DSP56001, DSP96002. Hardware in PTOLEMY refers to custom data paths and discrete (or glue) logic components in addition to the processor.

PTOLEMY's strength is its unified framework for the simulation of specifications as a set of heterogenous computation models. Specification in a particular model of computation is referred to as a design style that is encapsulated in objects called *domains*. A domain is comprised of blocks, targets and a scheduling discipline appropriate to its model of computation. In addition, *operational semantics* are embedded in blocks that govern their interaction with other blocks. Examples of supported domains are synchronous data flow (SDF), dynamic data flow (DDF), discrete event (DE) and signal-level digital hardware (Thor). A domain may embed another domain in its hierarchy. An embedded domain interacts with its parent domain by means of a procedures called the *event horizon*. Figure 12 explains the organization of domains and interface.

The event horizon is a key feature in PTOLEMY that makes it possible to interface event schedules from different domains. Domains can be classified into two categories: **timed** and **untimed** domains. A timed domain refers to a model of computation that produces events in the context of an associated time scale, for example, a discrete event domain. On the other hand, untimed domains do not have an absolute time association with their events, for example, data flow. When interfacing events across timed and untimed domains, there are several issues in event synchronization that must be worked out. In general, it would be hard, if not impossible, to provide a consistent simulation framework across concurrently independently active domains. However, due to embeddings

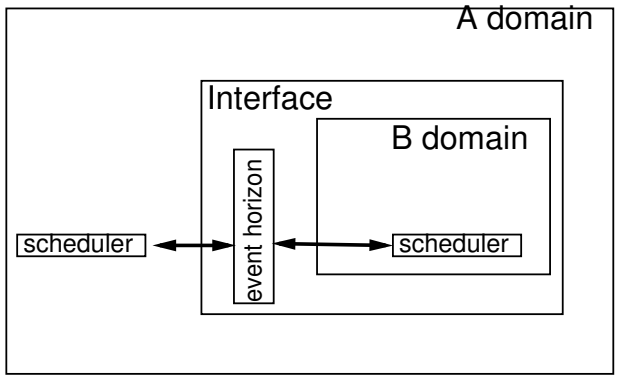


Figure 12: *Objects in* PTOLEMY

of these domains PTOLEMY makes it possible to carry out simulation under (conservative) restrictions. Stopping heuristics are used in domain simulations in order to make sure that inner timed domains do not temporally get ahead of the time in outer domains. Of course, inner untimed domains react in zero time. Outer untimed domains maintain timing attributes in order to set stop times for their inner domains. As an example, an event from an untimed domain causing an event into timed domain initiates a time scale on which to carry out further events in the timed domain until the inner domain has no more active events, thus making the timed domain appear like a functional block.

Despite the code generation abilities in its synchronous data flow (SDF) domain, PTOLEMY is primarily a simulation-oriented tool. Its specification language (for domains) is a procedural C++-type language. All models must be specified in this language which is extended to allow modeling of operators from various other languages (such as the '@' operator from Silage etc). Each model generates tokens. Models differ in values and timing interpretation of these tokens. Various models can be connected using a graphical schematic capture or a netlist language.

Even though semantically rich, PTOLEMY's syntax is awkward for specifying systems that are best captured in non-procedural languages. The use of a predefined library of a large number of models ameliorates this difficulty in specifying model functionalities.

The strength in heterogeneity by use of diverse computation models in PTOLEMY comes at the loss of an analytical handle on system properties. Further, it suffers in implementability of these models because of the necessity to specify these systems in

a simulation-oriented language which is not necessarily synthesizable. Nevertheless, PTOLEMY represents an important step towards simulation of complex systems. A PTOLEMY-like system that also allows heterogenous specification with associated synthesis tools (similar to an event scheduler) would be the next natural step toward creating a simulation and implementation framework.

2.1.2 CODES

Buchenrieder in [BV92] presents a framework for COncurrent DESign. The system is specified as a set of communicating parallel random access machines (PRAMs [HU79]). The design process is modeled using Petri nets. The emphasis here is on including both time-discrete and time-continuous behaviors in a single model. A component described using the RAM model is embedded in an I/O frame that defines its interaction with other models. The input specification can be simulated using Statemate[HLN+90] or System Description Language (SDL)[SSR89] tools. The synthesis into hardware relies on VHDL based synthesis tools.

The authors report successful design of an engine controller using the co-design methodology.

2.1.3 Rapid prototyping using SIERRA

Srivastava and Broderson [SB91] present a framework for rapid prototyping of systems that span across chips and multiprocessor boards in hardware as well as device drivers and operating system kernels in software. As opposed to PTOLEMY, the emphasis in SIERRA is on the implementability of the system. Due to the enormous complexity of the the systems represented, the analytical handle on system properties is further removed from achieved performance.

This work leverages the use of chip-level synthesis tools LAGER [SJR+91], HYPER [CPTR89], KAPPA [TRS+89] and DSP code synthesis tool GABRIEL [LHG+89] (GABRIEL functionalities were later incorporated in PTOLEMY) to present a framework for performing both activities. A system is specified as a network of concurrent sequential processes in VHDL. The communication between processes is by means of queues. This

specification is (manually) mapped into an *architecture template*. A mix of hardware and software tools and libraries are used to implement parts of the design. The main strength of this methodology lies in management of system complexity by using modularity and reusability afforded by existing libraries.

Using this methodology, the authors report a dramatic reduction in the overall design time to a matter of a couple of months. In addition, the framework affords the possibility of exploring design alternatives such as the effect of different processors and components. Successful designs of multi-board real time applications for a multi-sensory robot control system and for a speech-recognition system are reported [Sri92].

2.2 CAD for Hardware-Software Co-synthesis

2.2.1 COSYMA

CO-SYnthesis for eMbedded Architectures, COSYMA, performs partitioning of operations at the basic block level with the goal of providing speedup in program execution time using hardware co-processors. Figure 13 shows an overview of the system. Input to COSYMA consists of an annotated C-program [HE92]. This input is compiled into a set of basic blocks and corresponding DAG-based syntax graphs. The syntax graphs are helpful in performing data-flow analysis for definition and use of variables that helps in estimating communication overheads across hardware and software. The syntax graphs are partitioned using a simulated annealing algorithm under a cost function. This process is repeated using exact performance parameters from synthesis results for a given partition.

The partitioning task consists of the identification of the portions of the program that are suitable for synthesis into hardware in order to achieve a speedup in overall execution times. This partitioning, or hardware extraction is done by means of a simulated annealing algorithm using a cost function that yields potential speedup in execution times and reduction in communication overheads.

A timing constraint in COSYMA refers to a bound on the overall delay of a basic block. Since partitioning is done within a basic block, the timing performance of a

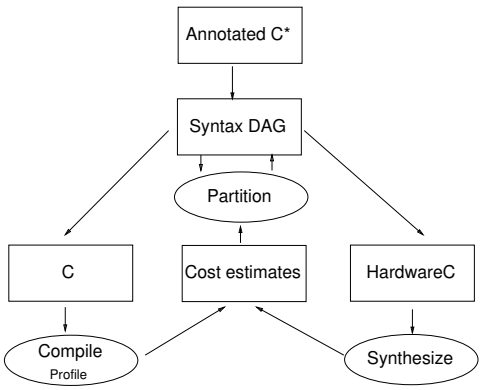


Figure 13: *Codesign flow in* COSYMA

hardware-software implementation is characterized by overall latency of the basic block. This latency includes delay overhead due to communication as the total number of variables that are alive across the partition boundary.

The chief advantage of this approach is the ability to utilize advanced software structures that result in enlarging the complexity of system designs. However, selective hardware extraction based on potential speedups makes this scheme relatively limited in exploiting potential use of hardware components. Further, the assumption that hardware and software components execute in an interleaved manner (and not concurrently) results in a system that under-utilizes its resources.

2.2.2 Use of non-deterministic finite state machines for co-design

Chiodo *et. al.* in [CGH+93a] present a formal model for specification of hardware software systems. The proposed model, Codesign Finite State Machines (CFSMs) is based on the theory of finite state machines. Figure 14 shows the overall flow for co-design. One of the important aspects of this approach to co-design is formal verification of the system design (not shown in the Figure).

The behavior of a system in this model is described by a 'trace' as a sequence of event instances. An event is defined by its name and the 'communication port' at which it occurs. A broadcast model of event communication is assumed. An instance of an event is different from the event itself and is identified by a time stamp at which the event

occurs. In general, events may carry values. Transitions are caused by trigger events, as opposed to pure value events which are used to select between transitions over the same set of trigger events based on data values. A CFSM is defined as a 5-tuple (I, E, O, R, F) as a set of input events (I), set of output events (O) with initial value (R), and a transition relation (F) from input to output events. E refers to a subset of events in I called trigger events. The state of CFSM is defined by the set of simultaneously occurring events that are both input (I) and output (O) events for the machine.

The reaction time to an input event can be (unbounded) non-zero. A CFSM contains both temporal non-determinism (unknown reaction times) as well as causal non-determinism since multiple input events may lead to the same output event (though an output event is generated by one and only one CFSM). The CFSMs are shown to be similar to classical finite state machines without an implied 'synchronous' hypothesis, which assumes that state transitions in a network of machines happen at the same time.

Hardware synthesis from CFSMs is performed by translating a CFSM into a network of (synchronous) Moore machines (with trace-equivalent/contained behavior) [CGH+93b] which are then synthesized using sequential logic synthesis algorithms implemented in SIS [SSM+92]. This translation is done assuming a finite reaction time to input events (i.e., no non-deterministic delay times are possible). This is accomplished by adding looping transitions on states to model the asynchronous nature of state transitions by synchronous machines[1]. The reaction to an event is present in a state immediate successor to state containing the event. Software synthesis is performed as translation of CFSMs into C-code blocks. The output events from CFSM are translated into communication events on virtual I/O ports.

The CFSM model is similar to other models based on communicating finite state machines, like SDL [SSR89] and CSIM [Sch90], though it lacks the storage extensions found in other models. However, for hardware-software co-design purposes, it is not the FSM nature of CFSM that is as important as its event-based model of communication. Its FSM nature does simplify the task of system verification. Thus, CFSMs are targeted for solving the system co-design and verification problems.

[1]This implementation, of course, assumes that the clock cycle time in synchronous implementation is much shorter than event interval times.

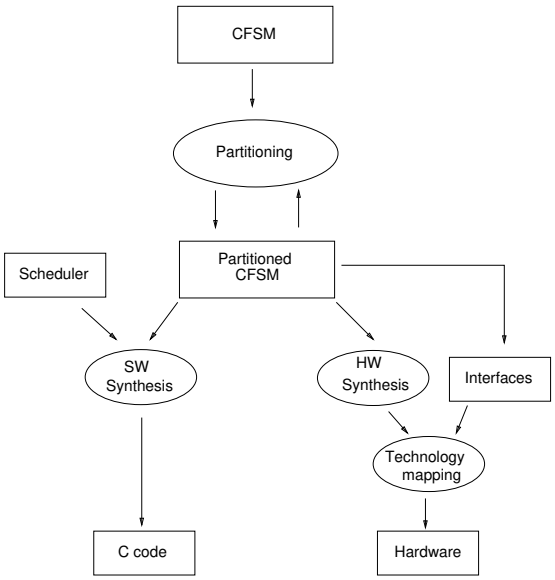


Figure 14: *Co-design from Finite State Machines*

The chief modeling limitation of this finite-state machine based approach to synthesis is that control and data operations are indistinguishable. Even though a BLIF-MV representation allows a concise representation of a data variable as a multi-valued logic variable, a particular value of a data variable defines a state. This is different from flow-graph based models where particular data values are inconsequential, and a system state is defined based on the state of control (for example, a particular path of execution). Because of this merging of control and data states in CFSMs, synthesis and optimization operations that are suited for data or control must either be applied uniformly or heuristics be used to determine a data state from a control state. Also, from a system design point of view, a CFSM based approach ignores modeling and the effect of timing constraints.

A CFSM based approach is expected to perform well for control dominated machines. However, for systems with a high-degree of data-intensive operations, a CFSM model may not be amenable to data-flow based optimizations.

2.2.3 Co-synthesis from UNITY

Barros, Rosenstiel and Xiong in [BRX93] present partitioning of system descriptions using the UNITY language. UNITY is a language for the specification of concurrent systems developed by Chandi and Misra [CM88]. A specification in UNITY consists of variable declarations and initializations followed by multiple-assignment statements. An assignment modifies a value held by a variable. This is referred to as a state transition in the execution of the UNITY program. Assignment can be composed in sequence or in parallel. In case of a choice, the selection of assignment statement to be executed is done non-deterministically.

The partitioning scheme presented classifies UNITY assignments according a set of five attributes which identify the degree of data dependency and parallelism between assignments. Associated with each of these attributes is a set of implementation alternatives. A reference implementation is chosen. A two-stage clustering algorithm then selects assignments to be grouped according to similarity of implementation alternatives, data dependencies, resource sharing and performance. The clustered assignments are scheduled for a given target architecture. Finally, an interface graph is constructed based on clustering results. This process is then reiterated based on satisfaction of design constraints.

2.2.4 Interface co-synthesis

Chou, Ortega and Borriello in [COB92] present an algorithm for synthesis of the interface between hardware-software systems. This interface allows interactions between the external devices and the program running on the processor. The result of interface synthesis is a software driver program and a logic circuit that provides a physical connection between the processor and external devices. This problem is solved in two parts: (a) allocation of physical ports on the processor to various devices; (b) selection of software driver routines.

Port allocation refers to assignment of processor ports to device ports. A processor port can be shared if its use by different device ports does not cause bus contention or a temporal overlap of the software drivers associated with the devices. Allocation of

processor ports to software function I/O calls to a device is performed in steps: attempt to share the device port to an already allocated port (conditional sharing); if conditional sharing fails then attempt to allocate a new processor port; if both these steps fail, then backtrack to find and make an allocated port shareable by addition of control hardware. If additional hardware does not help in sharing, an encoding transformation is applied to reduce I/O transfers. In the absence of applicability of any of these solutions, a memory-mapped I/O is selected which is always possible, though it comes with significantly higher delay and control overheads due to the protocols needed to implement memory operations over a shared communication medium.

The chief advantage of this approach is its considerable efficiency in building suitable input/output interfaces for controlling external devices. These are, as opposed to memory-mapped external communications, facilitated by a set of processor I/O ports. The processor I/O ports, though limited in number, provide a low overhead communication mechanism between software and hardware.

Chapter 3

System Modeling

This chapter examines issues in the specification and modeling of system functionality and constraints for systems that are target of hardware-software co-synthesis. The essential idea is to capture properties of a system without regard to its actual implementation. In practice it is hard to do, save for specific application domains. Some would argue that the more specific the application domain, the easier it is to develop a model. This work is targeted towards co-synthesis of embedded systems for which the following properties of target applications must be modeled and represented:

- The system consists of parts that operate at different speeds of execution,
- The interaction between parts of a system requires synchronization operations,
- There are constraints on the relative timing of operations.

A specification of a system functionality is done by means of a **language**. The language primitives and associated semantics determine the detailed functionality unambiguously. This degree of detail is often unnecessary for purposes of analysis. Hence **models** are needed.

In general, a model refers to an abstraction over its object, capturing important (but simple) relationships between important components of the object. Models are often needed in order to avoid creating detailed implementations. A model of a system helps

to estimate relevant properties, like area and delay, of its implementations.[1] Similarly, a *constraint model* is helpful in verifying satisfiability of imposed constraints.

For the purpose of model abstraction, sometimes generalizations and simplifications are made in order to represent objects that may be conceptually similar but differ in implementations. For example, a communication between two operations in a system model may be accomplished by means of a direct connection or over a shared medium such as memory or a bus or by using any of numerous protocols. A choice of a particular communication mechanism depends upon the individual operations and the part(s) to which they belong. For modeling purposes, a communication between two operations in the same part is abstracted as a dependency between the operations. Communication between operations belonging to different parts can be generalized to occur over *ports*. Ports represent communication to a shared-memory or inter-task communication by means of message-passing protocols (Section 3.4.1).

In the following section, we present our choice of the specification language. We then present a graph-based model and describe properties of the model used. Finally, we describe the constraints and a means of capturing them into the system model. Figure 15 shows the organization of this chapter.

3.1 System Specification using Procedural HDL

The search for a suitable language for specification is very much a subject of ongoing research. A detailed analysis of specification language issues is beyond the scope of this dissertation. For an overview of current research trends the reader is referred to [Har92] [Hal93] [Mic94], and [Sch92] [BW90b] for languages used for specifying real-time systems.

In order to formulate a practical co-synthesis approach, it is important that the language used have a developed path to hardware synthesis. From the point of view of hardware synthesis the most likely candidates are procedural and applicative languages [Joh83] [Sar89].

[1] Though sometimes *simulations* of an implementation are resorted to, particularly in cases where the mathematical complexity of a model analysis is overwhelming. A case in point are queueing systems. See Chapter 4.

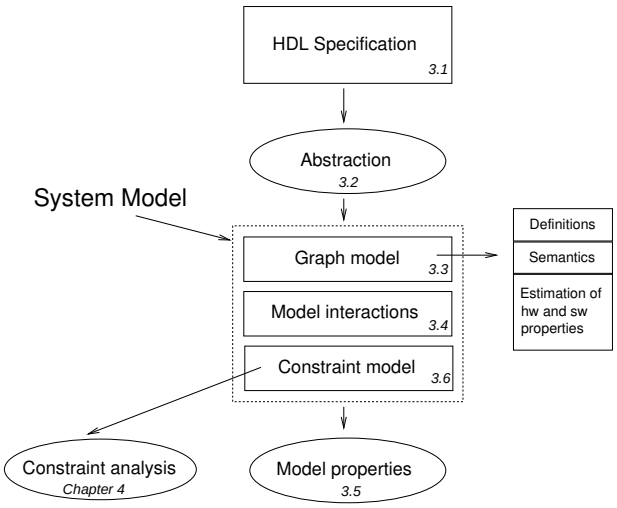


Figure 15: *Organization of Chapter 3*

The use of procedural hardware description languages (HDL) to specify system functionality for synthesis into digital hardware circuits has been gaining popularity in recent years. Most common languages used in practice today in this category are VHSIC Hardware Description Language (VHDL) [IEE87], and Verilog [TM91].

Part of the reason for the popularity of procedural languages in hardware specification is due to the familiarity of users with writing sequential programs. However, there are important differences in the expression of control in a program as opposed to its implementation in hardware. The program specification inherently assumes the existence of a single thread of control and static data storage, whereas the execution of operations in hardware is usually multi-threaded and is driven by the availability of appropriate data. Multi-threading is possible in hardware due to availability of multiple resources that are used to increase the degree of **concurrency** in operation execution. As a result, when describing hardware as a program, one is faced with the difficulty of specifying a concurrently executing set of operations as an ordered set of operations.

In contrast to an *instruction-driven* single-threaded linear-program representation, **data-flow graphs** (DFG) provide a *data-driven* representation that naturally models multiple-threads of execution (Figure 16). For this reason, the hardware for embedded controllers and non-recursive DSP algorithms is more appropriately represented by flow graphs instead of sequential programs used for procedural specification.

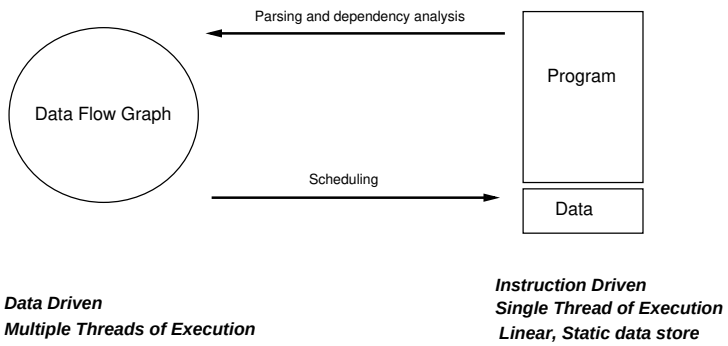


Figure 16: *Linear code versus data-flow graph representations*

To avoid this dichotomy of behavioral representations, most hardware synthesis algorithms operate on an *intermediate representation* based on data-flow graphs [McF78] [CKR84] [PPM86] [BCM+88]. This intermediate representation is generated by parsing and dependency analysis of the procedural input specification.

Data flow graphs have sufficient expressive power to represent either a hardware or software implementation. For example, a sequence of machine instructions can be represented by a machine-level data-flow graph. Indeed, the expression-evaluation trees generated by compilers (before the code-generation stage) and for hardware are forms of a data-flow graph. However, these data-flow graphs, consisting of operations described at the level of machine instructions, decrease the specification granularity too much to make them useful for the analysis needed for hardware and software co-synthesis. Therefore, data-flow graphs in our context are described using operations available at the language specification level.

Because of these strengths of a data-flow representation, we develop a system model based on data flow graphs. This model provides the basis for analyzing hardware and software implementations. From data-flow representations we can generate an equivalent sequence of instructions by scheduling various operation vertices in the data-flow graph. Operation scheduling techniques are important even in the case of a single thread of execution where static memory requirements are affected by scheduling, even though all schedules result in the same overall latency (see Chapter 5). Latency minimality in scheduling is realized by exploiting parallelism in the instruction stream which requires multiple execution threads. We consider the algorithms for evaluating data-flow graphs

and their equivalent linear-code representations in Chapter 5. Thus the ability to analyze and synthesize both hardware and software from data-flow graphs makes them a good candidate for an unified system model. This model is described in Section 3.2.

Specification. We specify system functionality in *HardwareC* [KM92a], a hardware description language. As mentioned before, the co-synthesis approach developed in this thesis is formulated on a system model based on data-flow graphs, and is independent of the actual language used to describe the system functionality. It is possible to use VHDL, Verilog or any other procedural HDL for system specification without altering the co-synthesis approach described in this dissertation. In the context of the present work, the choice of HardwareC is helpful in leveraging the existing path to hardware synthesis [MKMT90]. HardwareC follows much of the syntax and semantics of the programming language, C. Relevant features of the language are described in Appendix A. For further details the reader is referred to [KM90a].

The basic entity for specifying system behavior is a *process*. A process executes concurrently with other processes mentioned in the system specification. A process restarts itself on completion of the last operation in the process body. A process in HardwareC is similar to corresponding constructs in other hardware description languages. There are important differences, however. For example, in contrast to a process as a sequential set of operations in VHDL, a process in HardwareC can have nested sequential and parallel statement blocks. On the other hand, the synchronous semantics of HardwareC limit its expressiveness compared to VHDL. Example 3.1.1 describes a simple process specification.

Example 3.1.1. Example of a simple HDL process

```
process simple (a, b, c)
  in port a[8], b[8] ;
  out port c[8] ;
{
  boolean x[8], y[8], z[8] ;

  <
  x = read(a);
  y = read(b);
  >

  z = fun(x , y);
  write c = z;
}
```

This process performs two synchronous read operations in the same cycle, followed by a function evaluation and a write operation, then it restarts. □

Thus, the use of multiple processes to describe a system functionality abstracts the parts of a system implementation that operate at different speeds of execution. The effect of interaction between multiple processes is discussed further when we consider the system model in Section 3.2.

Memory and Communication

Communication refers to the process of transfer of information between operations. Some implementations of a communication require the execution of communicating operations at the same time. The process of bringing operation executions together is referred to as a **synchronization**. Synchronization is a general concept. Sometimes synchronization is needed to manage availability of shared resources. In our HDL specifications, synchronization is explicitly indicated only in the context of communication operations. A static resource allocation and binding paradigm is assumed, thus obviating the need for resource synchronization, i.e., avoiding conflicts when the same resource implements more than one operation. Therefore, synchronization in this work is mentioned in the context of communication operations.

All communication between operations within a process body is based on shared memory. This shared storage is declared as a part of the process body (for example variables `x`, `y` and `z` in the Example 3.1.1 above). Shared memory communication between operations is possible since it is relatively straight-forward to determine a (partial) ordering of operations within a given process body that ensures the integrity of memory shared between operations. However, the consistency of memory shared across concurrently executing processes must be ensured by the processes themselves. Inter-process communication is specified by message-passing operations that use a *blocking* protocol for synchronization purposes. As with shared memory variables, the only data-type available for a channel is a fixed-width bit-vector.

The use of message-passing operations simplifies the specification of inter-process communication. It should be noted, however, that it is easy to implement message-passing communication using memory shared between processes (the converse is not

true, however). Indeed, during system partitioning, reduction in communication overhead is realized by simplifying the inter-model communication as discussed in later sections. Example 3.1.2 below shows a process description containing a message-passing receive operation.

Example 3.1.2. Example of a process with unbounded delay operations

```
process example (a, b, c)
  in port a[8] ;
  in channel b[8] ;
  out port c ;
{
  boolean x[8], y[8], z[8] ;

  x = read(a);
  y = receive(b);
  if (x > y)
        z = x - y ;
  else
        z = x * y ;
  while (z >= 0) {
        write c = y ;
        z = z - 1 ; }
}
```

`read` refers to a synchronous port read operation that is executed unconditionally as a value assignment operation from the wire or register associated with the port `a`. `receive` is a message-passing based read operation where the channel `b` carries additional control signals that facilitate a *blocking* read operation based on the availability of data on channel `b`. □

3.2 System Model and its Representation

As mentioned earlier, a model refers to an abstraction of functionality over which the properties of an implementation can be explored. Due to the simplicity of models, these are extensively used in system analysis and synthesis procedures. A system model refers to a model of the complete system, whereas a process model refers to the abstraction of a process used in a system.

In order to correctly estimate properties of hardware and software in our target system implementation, we look for model characteristics that ease this process of estimation. Abstractions of operation-level concurrency and synchronization are important for hardware since these affect the *amount* of resources required for hardware implementations.

These also affect the satisfaction of timing constraints. Modeling software requires the abstraction of its interaction with a non-trivial runtime environment.

In the following we present a graph-based model that represents operation-level concurrency explicitly while making a provision for encapsulating operations due to the runtime system by a making suitable choice of additional source and sink operations in the graph-model.

As mentioned earlier, the specification of a digital system consists not only of a behavioral or algorithmic description of its functionality, but also of a description of its interaction with its environment and performance constraints. Correspondingly, any model of a digital system must also abstract these important components:

1. *Functionality or its behavior in response to environmental inputs.*

   Broadly speaking, there are two major way of modeling and analyzing the system behavior: algebraic process-based and graph-based. Algebraic modeling techniques such as process algebra [BK90, BW90a, Mil90] are commonly used in proof systems [AFR80, OG76]. Graph-based modeling uses techniques from graph theory to analyze system properties. The main difference between the two approaches is in the explicit expression of dependencies between processes and constituent operations. However, the equivalence between algebraic and graph-based modeling approaches has been demonstrated in [Tar81].

   We take a graph-based approach to system modeling and representation. Section 3.3 describes the model and its properties.

2. *A set of* ports *over which it interacts with its environment.*

   The behavior of an embedded system includes its interaction with the environment that influences current and future system behavior. This *reactive* nature of system functionality is expressed by means of its behavior on its ports. System interaction with an environment can be seen as a generalization of the interaction between its components. This generalization is supported by the port abstraction which in implementation can be a memory location, another system, or a device. Ports and port semantics are discussed in Section 3.4.1.

3. *Constraints on properties of its behavior.*

   Constraints are an important part of system specification. Constraints can be placed at various levels of abstraction. The specification of constraints is described in Section 3.6 and their analysis is presented in the Chapter 4.

3.3 The Flow Graph Model

The flow graph model captures the essential computational aspects of the target system. This model is presented in three parts: (1) Representation and definitions (Sections 3.3.1 and 3.3.2); (2) Execution semantics (Section 3.3.3); and (3) Abstraction of implementation attributes (Section 3.3.4).

3.3.1 Representation and definitions

Definition 3.1 *A* **flow graph model** *is a polar acyclic graph* $G = (V, E, \chi)$ *where* $V = \{v_0, v_1, \ldots, v_N\}$ *represent operations with* v_0 *and* v_N *being the source and sink operations respectively. The edge set,* $E = \{(v_i, v_j)\}$ *represents dependencies between operation vertices. Function* χ *associates a Boolean (enabling) expression with every edge. In the case of edges incident from a condition vertex or incident to a join vertex, the enabling expression refers to the condition under which the successor node for the edge is enabled.*

Table 1 lists operation vertices used in a flow graph model. It has been shown that this set of simple operations (that is, all operations except *wait* and *link*) provides a representation sufficient to capture universal computing power [Fos72]. A *wait* operation is needed to capture the timing uncertainty in the system behavior due to its reactive nature (see Section 3.4.2). The semantics of the *link* operation is discussed in the next section in the context of hierarchy in flow graphs. Note that the presence of multiple case values for the same branch leads to multiple edges between the condition and its successor vertex, thus making the flow graph a multigraph.

The flow graph model is similar to sequencing graph model by Ku [KM92a] with the following differences:

- A *wait* operation is added to abstract operations that represent synchronization events at model ports. This distinguishes a synchronization operation such as "`wait(signal)`" from a loop operation such as "`while(!signal)`" [2]. The reason for this distinction is that a software implementation of a wait operation is different from that of a loop operation. Whereas, due to the presence of multiple threads of execution in hardware, the wait operation is synthesized as a busy-waiting loop operation.

- Conditional `cond` and `join` operations. These operations have been added for the purposes of simplicity in data structures and constitute simple syntactic alteration to the sequencing graph model.

The advantage of the above changes to the sequencing graph models of [KM92a] is that they permit the distinction in abstraction of intra-model and inter-model communications as based on shared memory or message passing respectively. This issue is discussed in Section 3.5. However, the inclusion of conditional paths in the graph model introduces data-dependent execution paths of operations in addition to the possible data-dependent delay of operations. In contrast, the sequencing graph model features only the uncertainty due to data-dependent delay of operations by treating the conditional paths as separate graph models. Thus once invoked, *all* operations in a sequencing graph are eventually executed. As said earlier, this is only a syntactical alteration since invocation of operations on conditional paths is data-dependent in both cases.

For the sake of simplicity, a flow graph model, G, is often expressed as $G = (V, E)$. An edge $e_{ij} = (v_i, v_j) \in E(G)$ represents a dependency, $v_i > v_j$, between operations v_i and v_j such that for any execution of $G(V)$, operation v_i must always be initiated before operation v_j. An edge represents either a *data-dependency* $v_i \succ v_j$ or a control dependency $v_i \not\succ v_j$ between operations v_i and v_j. In the case of a data dependency, operation v_i produces (writes) data (variable) that is consumed (read) by the operation v_j. A control dependency from operation v_i to v_j indicates one of the following conditions:

1. operation v_j is environmentally constrained to be invoked only after invocation of v_i for correct behavior of the system being modeled,

[2] The loop operation is described later in the context of hierarchy.

Operation	*Description*
no-op	No operation
cond	Conditional fork
join	Conditional join
op-logic	Logical operations
op-arithmetic	Arithmetic operations
op-relational	Relational operations
op-io	I/O operations
wait	Wait on a signal variable
link	Hierarchical operations

Table 1: *Operation vertices in a flow graph*

2. v_j is invoked conditionally (based on output of v_i),

3. both operations v_i and v_j write to the same variable (multiple assignments).

4. operation v_j writes a variable that is read by operation v_i (anti-dependency).

In compiler parlance, a data-flow dependency is also called a read-after-write dependency. Note that in the last two cases (multiple assignments, and anti-dependency), dependencies occur only when the shared variable corresponds to a physical port. The relation $>^*$ indicates the transitive closure of the precedence relation $>$.

Note that any successor to a conditional operation is enabled if the result of condition evaluation selects the branch to which the operation belongs. This is expressed by the enabling condition associated with the edge from the condition vertex. In general, a multiple in-degree operation vertex is enabled by evaluating an *input expression* consisting of logical AND and OR operations over enabling expressions of its fanin edges. Similarly, on completion of an operation, all or a set of its successor vertices can be enabled. For each vertex, its *output expression* is an expression over enabling conditions of its fanout edges. These expressions determine the flow of control through the graph model.

A flow graph is considered **well-formed** if the input and output expressions use either AND or OR operations but not both in the same expression. For a well-formed graph, a set of input or output edges to a vertex is considered **conjoined** if the corresponding

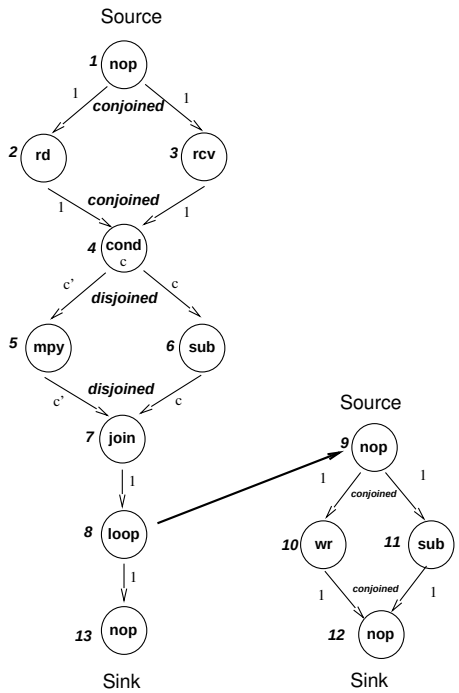


Figure 17: *Flow graph of process* `example`.

expression is a conjunction over inputs or outputs. Similarly, a set of edges is **disjoined** if the corresponding expression is a disjunction. A conjoined output directs the flow of control to all its branches, whereas a disjoined output selects one of the successors based on condition index. Similarly, a conjoined input requires arrival of control on all its inputs before enabling the vertex. Structurally this makes the flow graph a **bilogic** graph [Cer72]. For this reason, the flow graphs can be called **bilogic sequencing graphs** as opposed to (unilogic) **sequencing graphs** introduced in [KM92a]. Bilogic graphs are a fairly common occurrence in control graphs.

Example 3.3.3. Figure 17 shows example of a well-formed bilogic graph model for the process described in Example 3.1.2. The example shows a one-bit condition variable, `c = (x > y)`. In general, it can a multi-bit variable, thus leading to more than two branches. Note that for bilogic graphs, the `join` node is not essential since an appropriate input expression can be assigned to the successor node. However, a join node makes it easier in defining well-formed graphs. □

Overall, the flow graph model consists of concurrent data-flow sections which are ordered by control flow. The graph edges represent dependencies, while conjoined branches

Operation	*Description*	*Invocation times*
call	Procedural call	1
loop	Iteration	1 or constant > 1 or variable

Table 2: *Link vertices in a hierarchical flow graph*

indicate parallelism between operations. (Conjoined and disjoined fanin and fanout of a vertex are indicated by symbols '*' and '+' respectively).

3.3.2 Hierarchy

Flow graph models are hierarchically composed by means of **link** vertices. A link vertex represents a call to a flow graph model in the hierarchy. The called flow graph may be invoked one or many times depending upon the type of the link vertex. Table 2 lists types of link vertices and associated invocation times. Function and procedure calls are represented by a call link vertex where the body of function/procedure is captured in a separate graph model. A loop link operation consists of a loop condition operation that performs testing of the loop exit condition and a loop body. The loop body is represented as a separate graph model. All loop operations are assumed to be of the form

```
repeat {
    body
} until (condition);
```

that is, a loop body is executed at least once. HDL specification of 'while'-loops is transformed as follows:

```
while (condition) {          ⇒      if (condition) {
    body                                repeat {
};                                          body
                                        } until (! condition);
                                    }
```

A *system* consists of interacting parts, each of which can be abstracted into a flow graph model. A **system model** refers to the abstraction of the system. A system model consists of one or more flow graphs, that may be hierarchically linked to other flow

graphs. That is, a system model is expressed as, $\Phi = \{G_1^*, G_2^*, \ldots, G_n^*\}$, where G_i^*, represents the process graph model G_i and all the flow graphs that are hierarchically linked to G_i. Finally, a flow graph model that is common to two hierarchies of a system model is considered a **shared model** or a shared resource.

Example 3.3.4. Model hierarchy in an Error Correction System.

Figure 18 shows an error correction system (ECS) that models the transmission of digital data through a serial line. The ECS consists of an encoder front-end that reads input data word, encrypts and transmits it serially as an encoded stream into a noisy channel. The received bit stream is decoded and assembled into an output data word.

The system graph model consists of three process graph models: $\{G^*_{encode}, G^*_{decode}, G^*_{noise}\}$. The hierarchies of G^*_{encode} and G^*_{decode} share the flow graph model G_{xor}. □

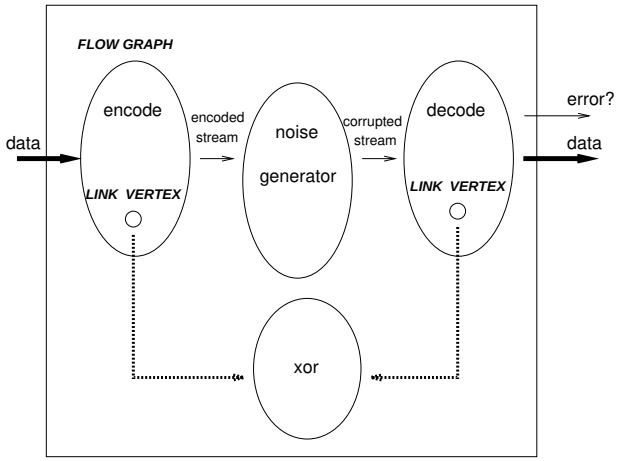


Figure 18: *Flow graph model for an error correction system.*

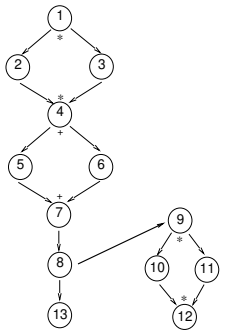
3.3.3 Execution semantics

In the previous section we showed the representation of the system functionality in a flow graph where operations represent vertices and edges represent dependencies between operations. This abstraction would be incomplete without an appropriate *operational semantics* associated with the execution of operations in a flow graph representation of the system model (also referred to as the flow graph model). An execution of the flow graph

model corresponds to a simulation of the system behavior. Thus, the development of an execution semantics is instrumental in understanding the possible behaviors exhibited by a system implementation. More importantly, it helps in reasoning about the validity of transformations to the graph model vis-a-vis system specification.

The execution semantics for a flow graph can be described as follows. At any time, an operation may be waiting for its execution, presently executing or having completed its execution. Correspondingly, we define the **state**, ς, of a vertex to be one of $\{s_r,\ s_e,\ s_d\}$ where s_r refers to the reset state, s_e to the enable state and s_d to the done state. An operation is enabled for execution once all its predecessors have completed execution in the case of a input-conjoined vertex; and once any of its predecessors has completed execution in the case of a input-disjoined vertex. The state of a vertex is changed from *done* to *reset* if all its successors are either reset or done. This semantics is general, and can support both pipelined and non-pipelined implementations of the graph model.

Example 3.3.5. Execution of process graph model `example` in Figure 17 reproduced below.



Non-pipelined

1	2	3	4	5	6	7	8	9	10	11	12	13
–	–	–	–	–	–	–	–	–	–	–	–	–
e	–	–	–	–	–	–	–	–	–	–	–	–
d	e	e	–	–	–	–	–	–	–	–	–	–
–	d	d	e	–	–	–	–	–	–	–	–	–
–	–	–	d	e	–	–	–	–	–	–	–	–
–	–	–	–	d	–	e	–	–	–	–	–	–
–	–	–	–	–	–	d	e	–	–	–	–	–
–	–	–	–	–	–	–	e	e	–	–	–	–
–	–	–	–	–	–	–	e	d	e	e	–	–
–	–	–	–	–	–	–	e	d	e	d	e	–
–	–	–	–	–	–	–	d	–	–	–	d	e
e	–	–	–	–	–	–	–	–	–	–	–	d
d	e	e	–	–	–	–	–	–	–	–	–	–
–	d	d	e	–	–	–	–	–	–	–	–	–
–	–	–	d	–	e	–	–	–	–	–	–	–
–	–	–	–	–	d	e	–	–	–	–	–	–
–	–	–	–	–	–	d	e	–	–	–	–	–
–	–	–	–	–	–	–	e	e	–	–	–	–
–	–	–	–	–	–	–	e	d	e	e	–	–

Pipelined

1	2	3	4	5	6	7	8	9	10	11	12	13
–	–	–	–	–	–	–	–	–	–	–	–	–
e	–	–	–	–	–	–	–	–	–	–	–	–
d	e	e	–	–	–	–	–	–	–	–	–	–
–	d	d	e	–	–	–	–	–	–	–	–	–
–	–	–	d	e	–	–	–	–	–	–	–	–
–	–	–	–	d	–	e	–	–	–	–	–	–
–	–	–	–	–	–	d	e	–	–	–	–	–
e	–	–	–	–	–	–	e	e	–	–	–	–
d	e	e	–	–	–	–	e	d	e	e	–	–
–	d	d	e	–	–	–	d	–	–	–	d	e
–	–	–	d	–	e	–	d	–	–	–	d	e
–	–	–	–	–	d	e	–	–	–	–	–	d
–	–	–	–	–	–	d	e	–	–	–	–	–

This table shows a particular sequence of operation executions for a given data input. Symbols ‘e’ and ‘d’ indicate enable and done states respectively. A dash ‘-’

indicates the reset state. No assumption about timing of the operations is made, that is, consecutive rows in the table above can be spaced arbitrarily over the time axis. Thus, the execution of a flow graph progresses as a **wavefront** of operations are enabled for execution. The operations may complete at different times depending upon the delay of the individual operations.

The table on the left shows the **non-pipelined** execution of the graph model, that is, the source vertex is enabled again only after the completion of all operations in the graph model. On the contrary, an execution is considered **pipelined** if the source operation is enabled before completion of all operations. Therefore, in a pipelined implementation, there is more than one wavefront of enabled operations that progresses through the graph model at any time. In general, pipelining of flow graphs requires generation of pipeline stall and bypass control needed to accommodate pipelining of variable delay and synchronization operations. In this work, we consider restricted pipelining using buffers only in the context of software synthesis in Chapter 4. For this pipelined execution, the minimum number of steps before the source operation can be enabled is determined by the maximum number of the steps taken by any operation. □

3.3.4 Implementation attributes

In this sub-section we define operation and graph attributes that are essential to performing the constraint analysis described in next chapter. Informally, an **implementation**, $\mathcal{I}(G)$, of a graph model, G refers to assignment of delays and size properties to operations in G, and a choice of *runtime scheduler*, Υ, that enables execution of source operations in G. This actual assignment of values is related to the hardware or software implementation of operations in G. For non-pipelined hardware implementations, the runtime-scheduler is trivial, the source operation is enabled once its sink operation completes (and the graph enabling condition is true for conditionally invoked graphs). For software, the runtime scheduler refers to the choice of a runtime system that provides the operating environment for execution of operations in G. A runtime system is characterized by its ability to preempt and prioritize operations. These are discussed in Chapter 5.

Size properties

Size attributes refer to the physical *size* and pinout of implementations of operations and graphs. The meaning of size for hardware and software implementations is different. A

hardware implementation consists of hardware resources (also called data-path resources), control logic, registers, and communication structures likes busses and multiplexor circuits. The size of a hardware implementation is expressed in units of gates or cells (using a specific library of gates) required to implement the hardware. Each hardware implementation has an associated *area* that is determined by the outcome of the physical design. We estimate hardware size assuming a proportional relationship between size and area. The size attribute for software consists of program and data storage required.

In general, it is a difficult problem to accurately estimate the size of the hardware required from flow graph models. Indeed, the size of implementation is one of the metrics that hardware synthesis attempts to minimize! Estimation in this context really refers to *relative* sizes for implementations of different flow graphs, rather than an absolute prediction of the size of the resulting hardware as formulated in [JMP89, KR93]. Notationally, the **hardware size**, S of an operation refers to its size as a sum of sizes of hardware resources required to implement the operation, associated control logic and storage registers. The size of a graph model is computed as a bottom-up sum of the size of its operations.

Even though we describe constraints later in this chapter, the effect of constraints on hardware size should also be noted. The effect of constraints, specifically on resource usage, is to limit the amount of available concurrency in the flow graph model. The more constraints on available hardware resources, the more operation dependencies are needed to ensure constraint satisfaction. The effect of timing constraints, on the other hand, is to explore alternative implementations at a given level of concurrency. Here we assume that the expressed concurrency in flow graph models can be supported by available hardware resources. That is, serialization required to meet hardware resource constraints has already been performed. This is not a strong assumption, since the availability of major resources like adders and multipliers are usually known in advance.

Capturing memory side-effects of a software implementation

A graph model captures the functionality of a system with respect to its behavior on its ports. The operational semantics of the graph model requires use of an *internal storage*

in order to facilitate multiple-assignments in HDL descriptions. Whereas additional variables can be created that avoid multiple assignments to the same variable, assignments to ports must still be multiply assigned in a flow graph model. Further, a port is often implemented as a specific memory location (that is, as a shared variable) in software. The memory side-effects created by graph models are captured by a set $M(G)$ of variables that are referenced by operations in a graph model, G. $M(G)$ is independent of the cycle-time of the clock used to implement the corresponding synchronous circuitry and does not include storage specific to structural implementations of G (for example, control latches).

The size, $S(G)$, of a software implementation consists of the program size and the static storage to hold variable values across machine operations. The static data storage can be in the form of specific memory locations or on-chip registers. This static storage is, in general, upper bounded by the size of variables in $M(G)$ defined above. In order to estimate software size, a flow graph model is not enough. In addition, knowledge of the processor to be used and the type of runtime system used would be needed. We discuss the processor abstraction and runtime environment in Chapter 5.

Pinout, $P(G)$ refers to the size of inputs and outputs in units of words or bits. A pinout does not necessarily imply the number of **ports** required. A pinout port may be bound to a number of input/output operations in a flow graph model.

Timing properties

The timing properties of the system model are derived from the timing properties of the flow graph models used to build the system model. For synthesis into hardware, the flow graph model is assumed to represent an abstraction of the synchronous digital hardware and as such its timing properties are derived using a *bottom-up* computation from individual operation delays.

Let us first consider non-hierarchical flow graphs, that is, graphs without link vertices. The **delay**, δ, of an operation refers to the execution delay of the operation. We assume that for a graph model, the delay of all operations are expressed as number of cycles for a given cycle time associated with the graph model. In a non-hierarchical flow graph, the delays of all operations (except `wait`) are fixed and independent of the input data.

The `wait` operation offers variable delay which may or may not be data-dependent depending upon its implementation. The **latency**, $\lambda(G)$, of a graph model, G, refers to the execution delay of G. The latency of a flow graph may be variable due to the presence of conditional paths.

Next, the hierarchical flow graphs also contain link vertices such as `call` and `loop` which point to flow graphs in the hierarchy. Therefore, an execution delay can be associated with link vertices as the latency of the corresponding graph model times the number of times the called graph is invoked. Since the latency can be variable, therefore, the delay of a link vertex can be variable. It may also be *unbounded* in case of loop vertices, since these can, in principle, be invoked unbounded number of times.

As mentioned earlier, the delay of `wait` operation depends upon its implementation. For instance, in a *busy-wait* implementation, the wait operation is implemented as a loop operation that iterates until the concerned input is received. This implementation is commonly used for hardware synthesis [KM92a]. Another implementation of wait operation would be to cause a *context-switch* which is particularly applicable for software implementations. For this implementation, the delay of the wait operation is characterized as a fixed quantity.

Length attribute and its computation

We define a lower bound on latency as the **length**, $\ell(G) \in Z^+$, of the longest path between the source and sink vertices assuming the loop index to be one for the loop operations[3]. In presence of conditional paths, the length is a vector, $\underline{\ell} = (\ell[i])$ where each element $\ell[i]$ indicates the execution delay of a path in G. The elements of $\underline{\ell}$ are the lengths of the longest paths that are mutually-exclusive. No particular ordering of elements in $\underline{\ell}$ is assumed.

The length computation for a flow graph proceeds by a bottom-up computation lengths from delays of individual operations. Given two operations, u and v with delays, δ_u, δ_v, these can be related in one of the following three ways in the flow graph:

Sequential composition: that is, $u > v$ or $v > u$. The combined delay of u and v is

[3]Recall that loop vertices represent 'repeat-until' type operations. The length computation treats the loop operation as a call operation.

represented by $\delta_u \odot \delta_v$ and is defined as

$$\delta_u \odot \delta_v \doteq \delta_u + \delta_v \tag{3.1}$$

Conjoined composition: when the operations u and v belong to two branches of a conjoined fork. A conjoined composition is denoted by $\otimes$ and the delay is defined as

$$\delta_u \otimes \delta_v \doteq \max(\delta_u,\ \delta_v) \tag{3.2}$$

Disjoined composition: when the operations u and v belong to two branches of a disjoined fork. This composition is denoted by symbol $\oplus$ and the combined delay is defined as

$$\delta_u \oplus \delta_v \doteq (\delta_u,\ \delta_v) \tag{3.3}$$

Clearly, a disjoined composition of two delays leads to a 2-tuple delay since the two operations belong to mutually exclusive paths. This composition of delays is generalized to composition of paths as follows. In case of a sequential composition of two path lengths, $\underline{\ell}_u$ and $\underline{\ell}_v$ with cardinality n and m respectively, the resulting path length contains $n \times m$ elements, consisting of sum over all possible pairs of elements of $\underline{\ell}_u$ and $\underline{\ell}_v$. In case of a conjoined composition, the resulting path length is of cardinality $n \times m$ and consists of maximum over all possible pairs of elements. Finally, in case of a disjoined composition, the resulting path length is of cardinality $n + m$ and contains all elements of $\underline{\ell}_u$ and $\underline{\ell}_v$. With this definition, the composition operators, $\odot, \otimes$ and $\oplus$ form a simple algebraic structure called commutative monoid, on the the power set of positive integers, Z^+ with 0 as an identity element.

In practice, one often needs only the upper and lower bounds on latencies. Notationally, ℓ_m and ℓ_M refer to the minimum and maximum element in $\underline{\ell}$ respectively. For well-formed graphs, ℓ_m and ℓ_M can be computed efficiently by collapsing conditional paths into a single operation vertex with minimum or maximum branch delay respectively. We state without proof the following properties:

$$\max(\underline{\ell}_1 \odot \underline{\ell}_2) = \ell_{1M} + \ell_{2M} \tag{3.4}$$

$$\min(\underline{\ell}_1 \odot \underline{\ell}_2) = \ell_{1m} + \ell_{2m} \tag{3.5}$$

$$\max(\underline{\ell}_1 \otimes \underline{\ell}_2) = \max(\ell_{1M}, \ell_{2M}) \quad (3.6)$$

$$\min(\underline{\ell}_1 \otimes \underline{\ell}_2) = \max(\ell_{1m}, \ell_{2m}) \quad (3.7)$$

$$\max(\underline{\ell}_1 \oplus \underline{\ell}_2) = \max(\ell_{1M}, \ell_{2M}) \quad (3.8)$$

$$\min(\underline{\ell}_1 \oplus \underline{\ell}_2) = \min(\ell_{1m}, \ell_{2m}) \quad (3.9)$$

Recall that a flow graph is considered unilogic if it all the fanin and fanout edges are only conjoined. A bilogic graph has both conjoined and disjoined relations on fanin and fanouts. The following theorem says that the maximum over the path length, $\underline{\ell}_M$, in a bilogic graph can be obtained simply by computing the longest path assuming the graph to be unilogic.

Theorem 3.1 *Given a bilogic graph, $G_{bilogic}$, let $G_{unilogic}$ be a graph created by treating all fanin and fanout edges to be only conjoined. Then,*

$$\ell_M(G_{bilogic}) = \ell(G_{unilogic}) \quad (3.10)$$

Proof: Proof is by induction over expression for path length. See Appendix B. ♯

Example 3.3.6. Latency and path length computations for bilogic flow graphs.

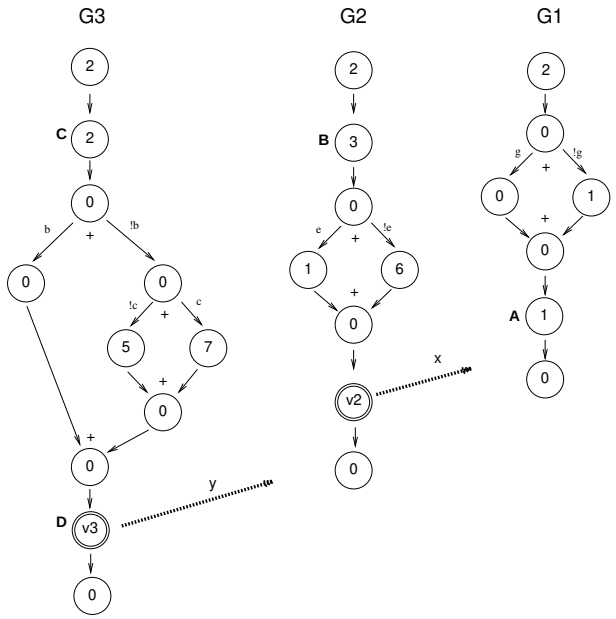
Figure below shows a process graph model, G_3 and graph models on its calling hierarchy. G_3 calls G_2 that constitutes body of a loop operation, v_3. G_2 in turn calls G_1 that constitutes the body of a loop operation, v_2. Numbers in the circle indicate delay of the operations.

For this set of graph models, the path lengths are:

$$\begin{aligned}
\underline{\ell}(G_1) &= 2 \odot (0,1) \odot (1) = (3,4) \\
\underline{\ell}(G_2) &= 2 \odot 3 \odot (1,6) \odot \underline{\ell}(G_1) = (6,11) \odot \underline{\ell}(G_1) = (9,10,14,15) \\
\underline{\ell}(G_3) &= 2 \odot 2 \odot (0,(5,7)) \odot \underline{\ell}(G_2) = 4 \odot (0,5,7) \odot \underline{\ell}(G_2) = (4,9,11) \odot \underline{\ell}(G_2) \\
&= (13,14,18,19,20,21,23,24,25,26)
\end{aligned}$$

Possible latencies are as follows:

$$\begin{aligned}
\lambda(G_1) = \underline{\ell}(G_1^*) &= (3,4) \\
\lambda(G_2) = \underline{\ell}(G_2^*) &= (6,11) \odot x \cdot (3,4) = (6+3x, 6+4x, 11+3x, 11+4x) \\
\lambda(G_3) = \underline{\ell}(G_3^*) &= (4,9,11) \odot y \cdot \lambda(G_2) = (4,9,11) \odot y[(6,11) \odot x(3,4)] \\
&= (4,9,11) \odot y(6,11) \odot yx(3,4) \\
&= (4+6y+3xy, 4+6y+4xy, 4+11y+3xy, 4+11y+4xy + \\
&\quad 9+6y+3xy, 9+6y+4xy, 9+11y+3xy, 9+11y+4xy + \\
&\quad 11+6y+3xy, 11+6y+4xy, 11+11y+3xy, 11+11y+4xy)
\end{aligned}$$



Note that the upper bound on latencies are given by

$$\begin{aligned}
\max \lambda(G_1) &= \ell_M(G_1) \\
\max \lambda(G_2) &= \ell_M(G_2) + (x-1) \cdot \ell_M(G_1) \\
\max \lambda(G_3) &= \ell_M(G_3) + (y-1) \cdot \ell_M(G_2) + (y-1) \cdot (x-1) \cdot \ell_M(G_1)
\end{aligned}$$

□

Rate of execution

The *instantaneous* **rate of execution**, $\widetilde{\rho}_i(t)$ of an operation v_i is the marginal number of executions n of operation v_i at any instant of time, t.

$$\widetilde{\rho}_i(t) \doteq \frac{dn}{dt} = \lim_{\Delta t \to 0} \frac{\Delta n}{\Delta t} \qquad (\text{sec}^{-1}) \tag{3.11}$$

Due to the discrete nature of executions (i.e., $n \in Z^+$), we define

$$\widetilde{\rho}_i(t) \doteq \begin{cases} \dfrac{1}{t - t_k(v_i)} & k \text{ such that } t_k(v_i) < t < t_{k+1}(v_i) \\ 0 & t \leq t_1(v_i) \end{cases}$$

where $t_k(v_i)$ refers to the start time of the kth execution of operation v_i. Assuming a synchronous execution model with cycle time τ, we define a *discrete rate of execution*

$$\widehat{\rho}_i(l) = \widetilde{\rho}_i(t) \mid_{t \,=\, l \,\cdot\, \tau} = \frac{1}{l - \frac{t_k(v_i)}{\tau}} \qquad (\text{cycle}^{-1}) \quad \frac{t_k(v_i)}{\tau} \leq l \leq \frac{t_{k+1}(v_i)}{\tau} \tag{3.12}$$

We define the rate of execution at invocation k of an operation v_i as the inverse of the time interval between its current and previous execution. That is,

$$\begin{aligned} \rho_i(k) &\doteq \widetilde{\rho}_i(t) \mid_{t = t_k(v_i)} \\ &= \frac{1}{t_k(v_i) - t_{k-1}(v_i)} \quad (\text{sec}^{-1}) \\ &= \frac{\tau}{t_k(v_i) - t_{k-1}(v_i)} \quad (\text{cycle}^{-1}) \end{aligned} \tag{3.13}$$

By convention, the instantaneous rate of execution is 0 at the first execution of an operation ($t_0 = -\infty$). Note that ρ_i is defined only at times when operation v_i is executed whereas $\widetilde{\rho}_i$ is a function of time and defined at all times. In statistics such a function is commonly referred to as of *lattice type*. In our treatment of execution rates and constraints on rates, only rates at times of operation execution are of interest. Hence we use the definition of ρ as the rate of execution.

Example 3.3.7. Figure 19 shows a simulation of the graph in Example 3.3.6.

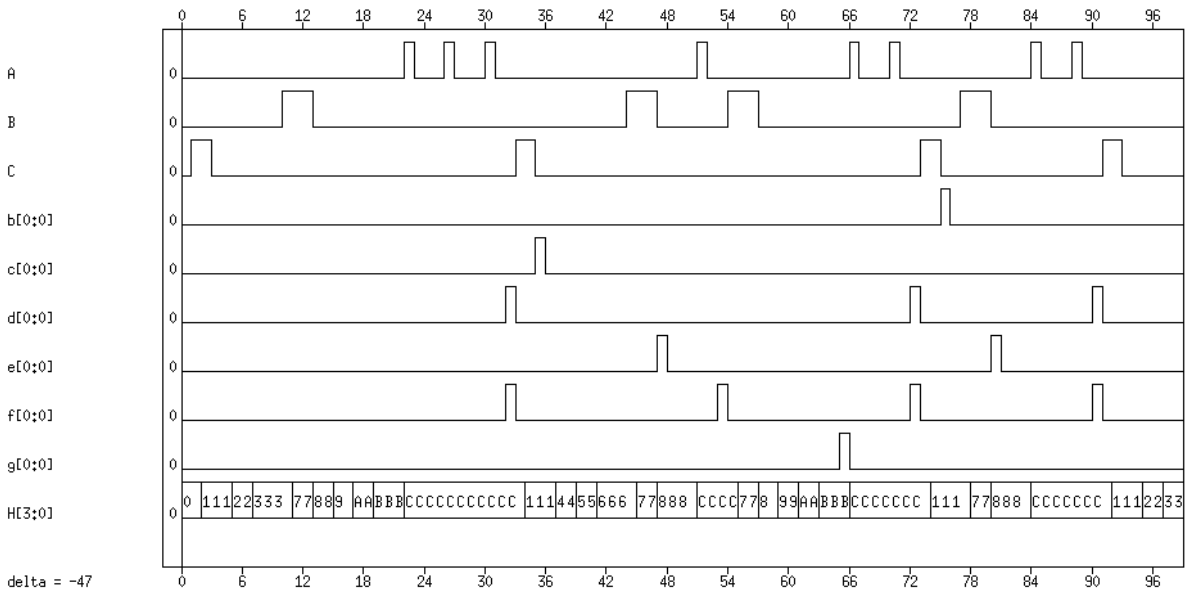


Figure 19: *Simulation of the graph model in Example 3.3.6*

Outputs labeled `A`, `B` and `C` refer to the execution of operation vertices 'A', 'B' and 'C' respectively.

From the figure, assuming the same cycle time for all graphs, the rate of execution of these operations is given as (cycle^{-1}) Consider operation 'A' that is executed at times 22, 26, 30, 51, 66, 70, 84 and 88. For this operation, $\rho_A(0) = 0$ by definition, $\rho_A(1) = \frac{1}{26-22} = \frac{1}{4}$ and so on. The following table lists the rate of execution for operations 'A', 'B' and 'C'.

$k \rightarrow$	1	2	3	4	5	6	7
$\rho_A(k)$	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{21}$	$\frac{1}{15}$	$\frac{1}{4}$	$\frac{1}{14}$
$\rho_B(k)$	0	$\frac{1}{34}$	$\frac{1}{10}$	$\frac{1}{23}$			
$\rho_C(k)$	0	$\frac{1}{32}$	$\frac{1}{40}$	$\frac{1}{18}$			

Thus the rate of execution of an operation varies as the interval between successive executions of the operation varies. A maximum rate of execution occurs following the shortest interval between two successive executions, and is always less than or equal to 1 cycle^{-1}. □

For a graph model, G, its **rate of reaction**, is defined as the rate of of execution of its source operation, that is,

$$\varrho_G(k) \doteq \rho_0(k) \tag{3.14}$$

The reaction rate is a property of the graph model and it is used to capture the effect on the runtime system and the type of implementation chosen for the graph model. To be specific, the choice of a non-pipelined implementation of G leads to

$$\varrho_G(k)^{-1} = \lambda_G(k) + \gamma_G(k) \tag{3.15}$$

where $\gamma(k)$ refers to the **overhead delay**, that represents the delay in reinvocation of G. $\gamma(k)$ may be a fixed delay representing the overhead due to a runtime scheduler or it may be a variable quantity representing delay in case of conditional invocation of G. For a pipelined implementation, the *degree* of pipelining determines the reaction rate of G. As the number of pipestages increases, the reaction rate of the graph model increases. With appropriate choice of pipeline buffers, it is possible to accommodate different rates of execution for operations in a graph model.

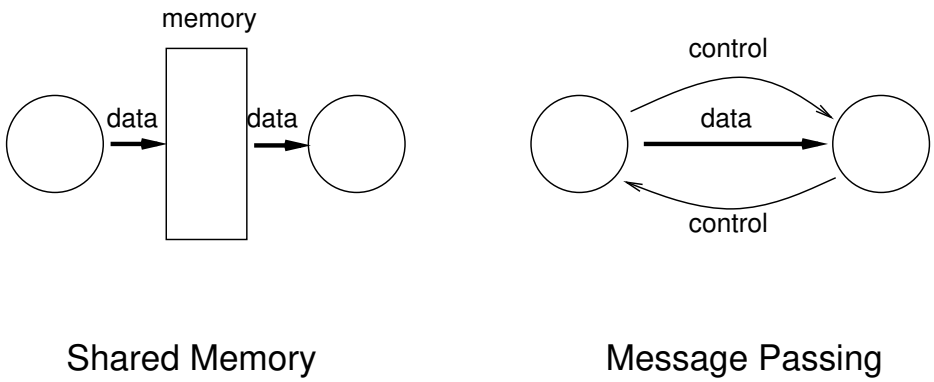


Figure 20: *Shared memory versus message passing communication*

3.4 Interaction Between System and its Environment

3.4.1 Ports and communication

Input and output operations in a digital system are performed over *ports* and *channels*. Since the HDL description can be implemented either as hardware or software, the semantics attached to the input/output ports of a system model must be suitable for specification of reactive hardware/software systems. In particular, the semantics of port/channel accesses must be compatible with the semantics of variable accesses. We examine these two together in the general context of communication in the graph model.

Communication in a system graph model Φ refers to the transfer of a data value from one operation vertex to another operation vertex or to the data transfer between operations and the environment external to Φ.

The operation vertex generating the data value is referred to as the producer vertex and the operation vertex using the data value is referred to as the consumer vertex. Communication between operations is either based on **shared memory** (SM) or **message passing** (MP). In the case of a shared memory communication between two operations, the sender operation modifies the contents of a storage (variable) that is shared by the receiving operation. A variable can be written by more than one operation. In the case of message passing communication between two operations, the actual data transfer is preceded by a **handshaked communication protocol** that requires the sending and

receiving operations to execute simultaneously (Figure 20). A communication protocol is a general term that encompasses many possible schemes to facilitate data transfer.

For all operations in a graph model, G, all communication is based on shared storage, $M(G)$. Inter-model communications are represented by explicit I/O operation vertices, which on execution, may alter the model storage $M(G)$. An I/O operation vertex encapsulates a sequence of operations as a particular 'communication protocol'. A communication protocol may be *blocking* or *non-blocking*. Since protocol is specified independently for the producer and consumer operations, it can be blocking or non-blocking at either or both ends. Further, a non-blocking protocol may also be finitely *buffered*. These protocols are built using simpler operations available in the flow graph model. Section 7.2.2 shows implementations of these protocols.

As mentioned earlier, dependencies between operations due to shared storage are represented by corresponding edges between operations. In the case of a read-after-write operation, a data-edge is indicated. For all other dependencies, a control edge is used. Communication based on message-passing is by means of *channels* that connect the communicating models. A channel, as a variable, can be accessed by multiple operations in the graph model. In our model, all communication dependencies are **statically** specified, that is, shared memory and message-passing channels are compiled from input descriptions.

3.4.2 Non-determinism in flow graph models

A flow graph model consists of operations that present fixed delay or variable delay during execution. This variance in delay is caused by the dependence of operation delay on either the *value* of input data or on the *timing* of input data.

Example of operations with value-dependent delays are loops with data-dependent iteration counts. Since the execution delay (or latency) of a bilogic flow graph can, in general, be data-dependent due to the presence of conditional paths, the delay of a call vertex is also variable and data-dependent. In bilogic flow graphs, link vertices present value-dependent delays.[4]

[4]We note here that for unilogic flow graphs, the conditional operation is also a link operation which presents a value-dependent delay corresponding to the delay of the operations in the branch taken.

The second category concerns operations with delays that depend upon a response from the environment. An operation presents a timing-dependent delay only if it has *blocking* semantics. The only operation in the flow graph model with blocking semantics is the *wait* operation. The *read* and *write* operations are treated as non-blocking. Their blocking versions are created by adding additional control signals and the wait operation. For this reason, the wait operation is also referred to as a *synchronization* operation.

Data-dependent loop and synchronization operations introduce uncertainty over the precise delay and order of operations in the system model. Due to concurrently operating flow graph models, these operations affect the order in which various operations are invoked. Due to this uncertainty, a system model containing these operations is called a **non-deterministic** [BEW88] model and operations with variable delays are termed **non-deterministic delay** or $\mathcal{ND}$ **operations**. Note that the non-determinism here is caused by the uncertainty in timing behavior of a concurrent system, and is different from the meaning of non-determinism used in the context of finite-state sequential machines [HU79].

3.5 $\mathcal{ND}$, Execution Rate and Communication

As mentioned earlier, a system model consists of parts (abstracted as graphs) that may execute at different speeds. For a given input/output operation, the system throughput at the corresponding port equals the rate of execution of the operation. For a flow graph containing no conditional and $\mathcal{ND}$ operations, the rate of execution of all operations is the same and is independent of input data. Therefore, the reaction rate of the graph, G,

$$\varrho_G(k) = \rho_{v_i}(k) \qquad \text{for all} \quad v_i \in V(G) \quad \text{and} \quad \text{for all} \quad k \geq 0$$

Thus the execution of G proceeds at a **single rate**. For a single-rate graph model, the system throughput at all ports is identical. For two single-rate graph models, G_1 and G_2, there exists a fixed number of invocations of G_1 with respect to an invocation of G_2 given by the ratio $\frac{\varrho_1}{\varrho_2}$.

Example 3.5.8. Single rate graph models.

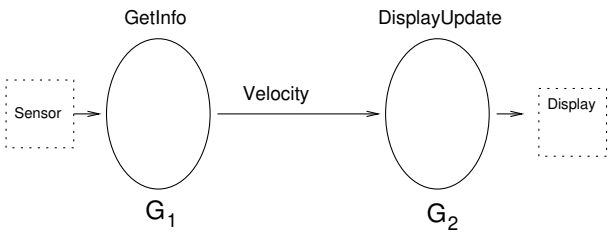


Figure above shows part of a vehicle cruise controller that consists of two single-rate graphs G_1 and G_2 implemented in hardware and software respectively. The latencies of the respective implementations are, $\lambda_{G_1} = 25 \cdot \tau_h$ sec with $\gamma_{G_1} = 0$ and $\lambda_{G_2} = 665 \cdot \tau_s \cdot$ sec with $\gamma_{G_2} = 85$ cycles. The clock cycle times for the hardware and software are 500 ns and 125 ns respectively. The reaction rates are $\varrho_{G_1} = 80,000/\text{sec}$ and $\varrho_{G_2} = 10,666.6/\text{sec}$. Therefore, for each execution of the software model, there are a fixed number of $\frac{80000}{10666.6} = 7.5$ executions of hardware. □

The reaction rate of a graph model containing conditional and $\mathcal{ND}$ operations is variable. A graph model with variable reaction rate is termed a **multi-rate** execution model. A multi-rate model has a bounded reaction rate if the model does not contain $\mathcal{ND}$ operations, else it is unbounded.

All communication in a single-rate graph model can be accomplished by means of shared storage since any execution of a graph model observes the partial order induced by its edges, *regardless of individual operation delays*. However, the relative ordering of operations *across* the graph model are dictated by the execution delay of individual operations. For software implementations this may lead to possible interleaving of operations in the graph models whereas a hardware implementation also includes the possibility of concurrent execution of operations across processes.

For any communication between operations across the graph models, a *safe* execution requires that the dependencies induced due to communication are always respected. For example, in Figure 21, a communication from operation c in G_2 to operation a in G_1 implies that only those executions are safe in which execution of c precedes execution of a. There are two ways to ensuring that this ordering from c to a is always observed. One is to construct a single flow graph model by merging G_1 and G_2 in which an edge is added from c to a. This may not always be possible, particularly, if G_1 and G_2

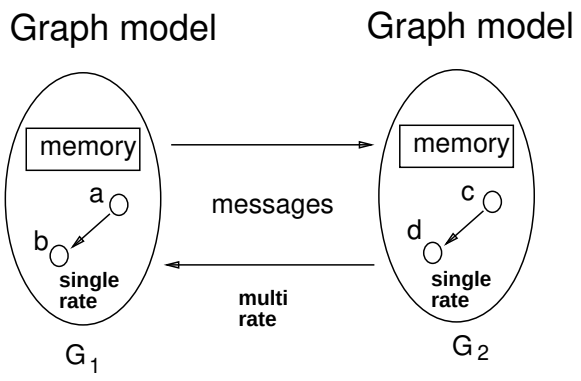


Figure 21: *Graph model properties*

have different reaction rates or use different clocks. An alternative is then to make the operation a block until c is available (and vice-versa). This is accomplished by using a message-passing protocol between G_1 and G_2.

Example 3.5.9. Use of message passing for communication across two graph models.

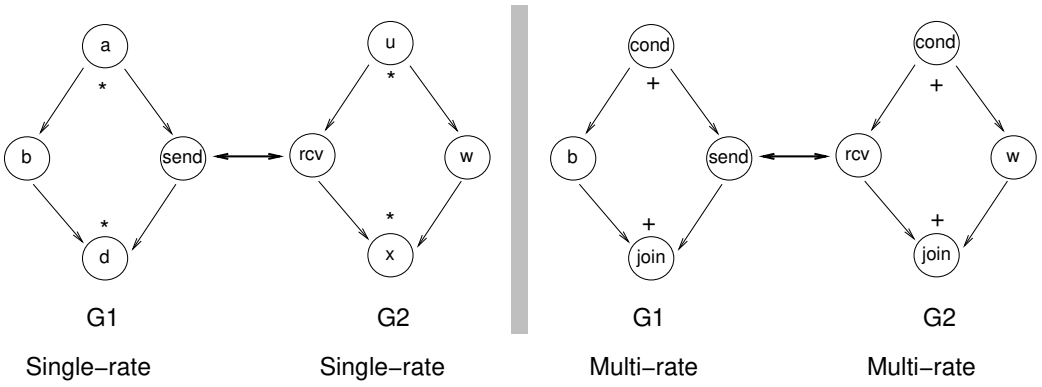


Figure 22: *Communication across models.*

Figure 22 shows communication across two models, G_1 and G_2. In the first case, G_1 and G_2, are single rate. Since the `send` and `receive` operations are invoked for each execution of the respective graph models, therefore, the rates of execution of operations `b` and `w` are identical. In the second case the execution of synchronization is conditionally invoked. Hardware-software implementations of G_1 and G_2 benefit by this synchronization operation since it allows G_1 and G_2 to run at their reaction rates and synchronize only when a communication is indicated.

The advantage of message-passing is realized when communicating across multi-rate model(s). In the case of single-rate models, use of message passing provides a notational simplicity; however, it is more efficient to implement the communication based on shared-memory. This is because a shared memory communication uses much less overhead both in operation delay as well as control complexity. In this context, a completely non-blocking message-passing communication can be thought of as a shared memory communication. □

In principle, communication between two single-rate models can be accomplished by means of shared memory. This is, however, not convenient for different implementations, such as one in hardware and the other in software, of single-rate graph models, even if they have the same reaction rate. This is because, hardware and software implement storage differently even though the access semantics in graph models are identical. On the other hand, communication across two multi-rate models using the same implementation can be accomplished by shared memory, as shown by the following example.

Example 3.5.10. Implementation of communication across multi-rate models by means of shared memory.

An example of a multi-rate model using shared memory is the complete hardware implementation of the loop operation in [FKD92]. As shown in Figure 23, the loop

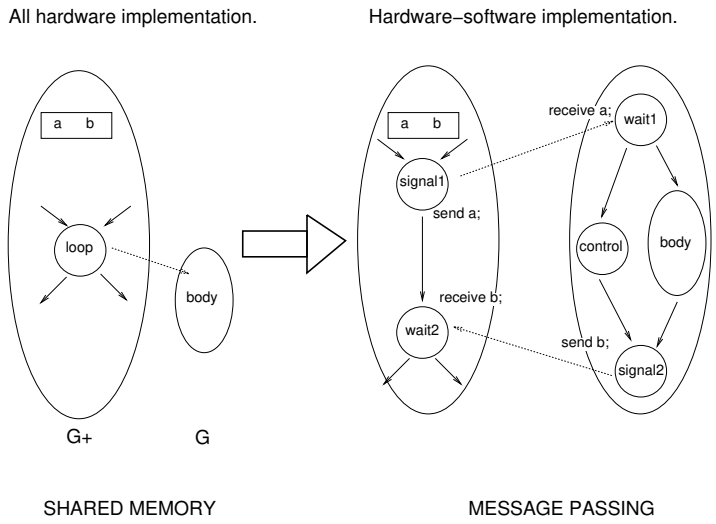


Figure 23: *Shared-memory versus message-passing implementations of loop operation.*

body communicates with the calling body by means of shared storage, even though the operations in the loop body are executed multiple times for each execution of the

operations in the calling-body. The safety of the executions is ensured by blocking the execution of the parent body (by operation wait2) until the loop has terminated. □

A question naturally arises: what then is gained by using message passing based communication between the parent body and the loop body? The answer depends upon the memory side-effects created by the loop operation and the kind of loop operation being used. In the case of loop operations with no memory side-effects, the parent body need not be blocked, i.e., operation `wait2` can be eliminated, thus providing additional flexibility in scheduling of operations. While such flexibility comes with significant additional overhead in hardware, software implementations of flow models require relatively minor operations in the run-time environment to make use of this flexibility to overlap operation. In particular, the execution of a blocking wait operation can be overlapped to that of run-time scheduling operation. This is accomplished by including the delay of the `wait1` operation into the runtime delay without leading to loss of processor utilization in software. Therefore, different implementations of operations even within a graph model are more conveniently handled by the use of message-passing based communications than shared memory. This allows the hardware and software parts to run at suitable reaction rates while synchronizing only when necessary.

3.6 Constraints

Constraints are an integral part of the system specification. Constraints can be on the size and/or on the performance of the desired implementation. Typically, the goal of system synthesis is to explore solutions that optimize parameters while observing constraints on specified parameters.

In the context of mixed system synthesis, performance constraints can be on system response time and degree of resource utilization. These performance constraints have varying degrees of time granularity and tolerances. Further, different embedded systems operate under varying types and degrees of constraints. For example, for systems used in control applications, real-time response time constraints are most important, while to systems used in on-line transactions and data-processing, synchronization and consistency

constraints are of most importance.

In general, these performance constraints are too abstract to be handled directly on a system model that is described at the level of individual operations. For this reason, we first devise timing constraints that apply to the level of individual operations, develop a runtime system to support the operation of mixed systems and finally develop a relationship of operation-level timing schema to system performance parameters in the context of the runtime system environment. For a given set of performance constraints, there can be more than one assignment of operation-level constraints. The selection of appropriate operation-level constraints corresponding to a given performance constraint would then require a clear understanding of the effect of constraints in system partitioning for hardware and software. In the following, we discuss operation level constraints and their implications for a mixed system design. The system performance constraints are developed in the context of a runtime system described in Chapter 5.

Timing Constraints are of crucial importance, since in our approach to co-synthesis they determine the feasibility of mixed implementations. We use these constraints to drive a partitioning algorithm in choosing operations for hardware or software implementation. Timing constraints are of the following types:

1. Min/max delay constraints

2. Execution rate constraints

These operation-level constraints are devised in order to make the task of constraint analysis tractable in the context of our system model based on flow graphs. We note that these two constraint types capture the durational and deadline timing constraints used in specifying real-time systems [Das85].

3.6.1 Min/max delay constraints

Let us first consider the timing constraints of the first type, that is, the min/max delay constraints. Min/max constraints are specified operation-to-operation and are needed to ensure required separation between the execution of two operations. If the operations are

input/output related, then a min/max constraint implies a certain bound on the response from the environment in which a circuit operates (e.g., memory), or a bound on the response of the system to the environment.

In the case where min/max constraints are on operations other than input/output, min/max constraints account for delays of specific components or resources. By default, any sequencing dependency between two operations, induces a minimum timing constraint which must be satisfied in order to observe the execution semantics of the flow graph.

For operations implemented in software that is running on a single processor, the important min/max constraints are those on input/output operations. This is due to the fact that a single-processor enforces execution of only a single thread of control at any time. In fact, the insensitivity to relative inter-operation delays in interleaved execution threads is a necessary condition for ensuring functional correctness of the software. Note this situation is different in hardware (or software on multiple processors) where multiple threads of control can coexist simultaneously.

Recall that $t_k(v_i)$ represents the *start time* of the kth occurrence of operation v_i. A *minimum timing constraint*, $l_{ij} \geq 0$ from operation vertex v_i to v_j is defined by the following relation between the start times of the respective vertices:

$$t_k(v_j) \geq t_k(v_i) + l_{ij} \qquad \text{for all} \quad k > 0 \tag{3.16}$$

For notational simplicity, we drop the suffix k when the constraint applies universally to k. Similarly a *maximum timing constraint*, $u_{ij} \geq 0$ from v_i to v_j is defined by the following inequality:

$$t(v_j) \leq t(v_i) + u_{ij} \tag{3.17}$$

3.6.2 Execution rate constraints

Execution rate constraints refer to constraints on the interval of time between successive executions of the *same* operation. In particular, execution rate constraints on input (output) operations refer to the rates at which the data is required to be consumed (produced). We assume that each execution of an input (output) operation consumes (produces) a *sample* of data. Execution rate constraints on input/output operations are referred to as *data*

rate constraints. In the literature, data rate constraints have also been referred to as *throughput* constraints as opposed to min/max constraints which are expressed on delays associated with a single execution.

A *minimum data rate constraint*, r_i (cycles^{-1}), on an input/output operation defines the lower bound on the instantaneous execution rate of operation v_i. Similarly, a *maximum data rate constraint*, R_i (cycles^{-1}), on an I/O operation defines the upper bound on the instantaneous execution rate of operation v_i.

$$\begin{aligned} \rho_{v_i}(k) &\leq R_i & \forall\ k > 0 \qquad & [\text{max rate}] \\ \Rightarrow\ t_k(v_i) - t_{k-1}(v_i) &\geq \tau \cdot {R_i}^{-1} & \forall\ k > 0 & \end{aligned} \tag{3.18}$$

Similarly,

$$\begin{aligned} \rho_{v_i}(k) &\geq r_i & \forall\ k > 0 \qquad & [\text{min rate}] \\ \Rightarrow\ t_k(v_i) - t_{k-1}(v_i) &\leq \tau \cdot {r_i}^{-1} & \forall\ k > 0 & \end{aligned} \tag{3.19}$$

Let us now consider, an operation, v_i, in a graph model G. In general, when considering rate of execution of v_i we must consider the successive executions of v_i that may belong to separate invocations of G. On the other hand, a **relative execution rate constraint** of an operation, v_i, with respect to a graph model, G, is a constraint on the rate of execution of v_i *when G is continuously enabled and executing*. In other words,

$$r_i^G \leq \rho_{v_i}(k) \leq R_i^G \tag{3.20}$$

for all $k > 0$ *and* there exists an execution, j, of G such that

$$t_j(v_0(G)) \leq t_{k-1}(v_i) \leq t_k(v_i) \leq t_j(v_N(G)) \tag{3.21}$$

The motivation behind the relative rate of execution is to express rate constraints that are applicable to a specific *context* of execution as expressed by the flow of execution that enables the specified graph G. Clearly, a relative rate constraint is meaningful when expressed relative to a flow graph in the hierarchy in which the operation resides. Further, as we shall see in the following chapter, the maximum execution rate of an operation is achieved when the flow graph in which the operation belongs is continuously enabled. Therefore, a relative maximum rate constraint, R^G, is always trivially satisfied if a corresponding maximum rate constraint is satisfied. Therefore, it is the relative minimum rate constraints that are used in practice.

3.6.3 Specification of timing constraints

Operations in the flow graph model correspond to language-level operations, that is, operations supported in the HDL. Therefore, it is easy to specify timing constraints by tagging the corresponding statements in HDL descriptions.

In the case of nested loop operations, rate constraints are indexed by the corresponding loop operations. The loops are indexed by increasing integer numbers. The inner-most loop is indexed 0. In the Example 3.6.11 below there are two relative rate constraints on the read operation with respect to the two while statements.

Example 3.6.11. Specification of rate constraints in presence of nested loop operations.

```
process example (frameEN, bitEN, bit, word)
   in port frameEN, bitEN, bit;
   out port word[8];
{
   boolean store[8], temp;
   tag A;

   while (frameEN)
   {
       while (bitEN)
       {
A:             temp = read(bit);
               store[7:0] = store[6:0] @ temp;
       }
       write word = store;
    }
    attribute "constraint minrate of A   = 100 cycles/sample";
    attribute "constraint minrate 0 of A =   1 cycles/sample";
    attribute "constraint minrate 1 of A =  10 cycles/sample";
}
```

In this example, an r of 0.01 per cycle is indicated on the read operation. In addition, two *relative* minimum data rates of 1 and 0.1 per cycle are indicated for the read operation with respect to loops `while(bitEN)` and `while(frameEN)` respectively. □

3.7 Summary

The design of a suitable language for system specification is a topic of active research interest and beyond the scope of this work. We follow the current practice in system design by using a procedural language input as specification. Clearly, this does not imply

that this means of specification is ideal, except that it serves our purpose of specifying sufficient information in order to make system co-synthesis possible.

Most of this chapter defines the input model used for abstraction of system functionality and constraints. We develop a representation based on graphs that meets the essential requirements of capturing explicit concurrency, synchronization, data and control flow. This model is general enough to allow synthesis of both hardware and software as well as their pipelined or non-pipelined implementations. The properties of implementations are captured as attributes of the graph model. Due to this abstraction of an input description into operations and dependencies, it is argued that different specification languages can be used without altering the co-synthesis paradigm. The scope of systems handled by the co-synthesis approach is succinctly defined by this abstraction. HardwareC or any other language (not necessarily procedural) can be used to describe the system functionality for this purpose. (The converse is of course not true. That is, not all VHDL or C descriptions can be synthesized into hardware-software as described here.)

The flow graph model consists of a graphical structure and an execution semantics that formally abstracts HDL descriptions. Computationally, the flow graphs are similar to control graphs. However, unlike the general control-data-flow graph models, the flow graph model also captures the memory side-effects of an implementation by means of a set of variables that are associated with a graph model. Most representations based on data-flow graphs, disallow this multiple assignment, thus creating a correspondence between a single data-item to each edge in the graph. Our motivation for multiple assignments stems from our need to treat all communication, whether by ports in hardware implementations or by means of storage in software implementations, symmetrically. Therefore, variables, in general, can be multiply assigned, similar to ports. Indeed a port can itself be implemented as a memory location in case of memory-mapped I/O operations.

The distinction between rate of execution of an operation as it relates to the structure of the flow graph in which the operation belongs helps in analysis of constraints on rates of execution by propagating known rates of execution through the graph model. By definition, in a single rate graph model, all operations execute at the same rate. When interfacing two graphs models (with possibly different implementations), their rates of

reaction are important in selecting the protocol for communication across the models. The protocols can be blocking, non-blocking or buffered. Since these protocols have different implementation costs, analysis of reaction rates can be used to minimize communication costs (for example, by eliminating redundant synchronizations).

Chapter 4

Constraint Analysis

In this chapter, we present timing constraint abstraction and analysis techniques. The primary objective of constraint analysis is to examine the mutual-consistency of timing constraints, and to answer the question about the existence of a system implementation that would satisfy the timing constraints. This analysis assumes that any constraints on availability of hardware resources have already been resolved as additional control dependencies in the flow graph model. Therefore, the available concurrency in the flow graph model can indeed be supported by the available hardware.

For each invocation of a flow graph model, an operation is invoked zero, one or many times depending upon its position on the hierarchy of the flow graph model. The execution times of an operation are determined by two separate mechanisms:

- The runtime scheduler, Υ
- The operation scheduler, Ω

The runtime scheduler determines the invocation times of flow graphs, which may be as simple as fixed-ordered where the selection is made by a predefined order (most likely by the system control flow). This is typically the case in hardware implementations where the graph invocation is purely a subject of system control flow. Software implementations of the runtime scheduler tend to be more sophisticated, due to the ease in altering system control flow. The runtime scheduling is also referred to as long term scheduling (as opposed to short term operation scheduling performed by the operation scheduler). We

make a distinction between two major types of runtime environments: non-prioritized and prioritized. A prioritized environment assumes an ordering of graphs irrespective of system control flow. The effect of a runtime scheduler is presented in the next chapter.

4.1 Scheduling of Operations

Given a graph model, $G = (V, E)$, the selection of a **schedule** refers to the choice of a function, Ω that determines the start time of the operations such that graph execution semantics shown by the following equation:

$$t_k(v_i) \geq \max_{j \ni v_j > v_i} [t_k(v_j) + \delta(v_j)] \tag{4.22}$$

is satisfied for each invocation $k > 0$ of operations v_i and v_j. Here $\delta(\cdot)$ refers to the delay function and returns the execution delay of the operation.

Given a scheduling function, a timing constraint is considered satisfied if the operation initiation times determined from applying the scheduling function satisfy the corresponding Inequalities (3.16, 3.17, 3.18 or 3.19). Clearly, the satisfaction of timing constraints is related to the choice of the scheduling function. Before proceeding to analyze satisfiability of timing constraints, let us take a look at different *types* of scheduling functions that can be applied to the flow graph model described in the previous chapter.

We consider first a model, G, where the delay of all operations in G is known and bounded. A schedule of G maps vertices to integer labels that define the start time of corresponding operations, that is, $\Omega_s : V \mapsto Z^+$ such that operation start times, $t_k(v_i) = \Omega_s(v_i)$ satisfy Inequality 4.22. A schedule is considered minimum if $|t_k(v_i) - t_k(v_o)|$ is minimum for all $v_i \in V$. For each invocation of G, since the start times of all operations are fixed for all executions of G (that is, for all k), such a schedule is referred to as a **static schedule**. Various static scheduling disciplines are possible, for example, As Soon As Possible (ASAP), As Late As Possible (ALAP), List or Force Directed Scheduling (refer to Chapter 5 in [Mic94] for an overview of static scheduling methods).

All static scheduling disciplines require the determination of fixed and known delays for all operations. In the presence of conditional, loop and wait operations, not all delays can be fixed or known statically, thus making a determination of an unique operation

start time impossible. This provides the motivation for a scheduling function that does not require $\delta(\cdot)$ to be a fixed quantity. We consider one such function, called the relative schedule [KM92b], which uses runtime information to determine operation start times for each invocation of a graph model.

A **relative schedule** function maps vertices to a *set* of integers representing *offsets*. An offset $\theta_{v_j}(v_i)$ of vertex v_i with respect to vertex v_j is defined as the delay in starting execution of v_i after completion of operation v_j. Offsets are determined relative to vertices which the execution of v_i (transitively) depends upon. That is,

$$t_k(v_i) \geq t_k(v_j) + \delta(v_j) + \theta_{v_j}(v_i) \quad \text{if} \quad v_j >^* v_i$$

For a given vertex, v_i a set, $\mathcal{A}(v_i)$ of *anchor* vertices is defined as the set of conditional ($\mathcal{CD}$) and loop, wait ($\mathcal{ND}$) vertices that have a path to v_i:

$$\mathcal{A}(v_i) = \{v_j \in V : v_j >^* v_i,\ v_j \text{ is } \mathcal{ND} \text{ or } \mathcal{CD}\} \tag{4.23}$$

A relative schedule function, Ω_r is defined as a set of offsets for each operation such that operation start time satisfies the following inequality:

$$t_k(v_i) \geq \max_{a \in \mathcal{A}(v_i)} [t_k(a) + \delta(a) + \theta_a(v_i)] \tag{4.24}$$

Since the quantity $\delta(a)$ is known only at runtime, the operation start time under relative schedule is determined only at the runtime.

Inequality 4.24 can be derived from the inequality 4.22 by expressing the latter over the transitive closure, $G^{>*}$, of G and then adding the known operation delays, δ, as offsets from unknown delay operations. Recall, that a transitive closure of a graph refers to a graph with edges indicating direct or transitive dependency between operations. Clearly, a solution to Inequality 4.24 will also satisfy Inequality 4.22 if the offsets, $\theta_{v_j}(v_i) \geq \ell(v_j,\ v_i)$, where $\ell(v_j, v_i)$ refers to the path length from vertex v_j to vertex v_i. Finally, a relative schedule is minimum if it leads to minimum values of all offsets for all vertices.

One of the interesting properties of a relative schedule is that it attempts to express the (spatial) uncertainty associated with conditional invocations of an operation ($\mathcal{CD}$) as its temporal uncertainty by treating it as an unbounded delay ($\mathcal{ND}$) operation. Thus, a

conditional operation is same as an data-dependent loop operation where operations on its branches are invoked a variable number of times (0 or 1) depending upon data values. For the purposes of relative scheduling, variable delay operations are treated as unknown delay operations in [KM90c]. Due to this treatment, the corresponding flow graph used for relative scheduling is **unilogic**, since conditional branches belong to separate graphs same as in the case of loops. Of course, this idea can be carried further by treating all operations as unbounded delay operations and computing the start times of operations at runtime. Such an implementation of a flow graph would be similar in architecture to data flow machines [Wat84]. In terms of the latency of execution, such a dynamic scheduler will give the most 'compact' schedule. There is, however, an overhead cost of control associated with increasing the number of unbounded delay operations that makes such an architecture unsuitable for either gate-level hardware or software on conventional general-purpose processors. Hence we seek to minimize the number of unknown delay operations in the graph model. Filo *et. al.* in [FKD92] address the problem of minimization of the number of unbounded delay operations that belong to the anchor set of an operation based on the notion of irredundant anchor operations that are essential in determination of the start time of an operation. This process can be complemented by taking out as many operations out of the scope of unbounded delay operations as possible.

In this context, bilogic flow graphs treat conditional operations not as unknown delay operations, but as variable and bounded delay operations. Correspondingly, we develop a bilogic relative schedule that uses bounds on the variable delay operations to develop a schedule. Depending upon the actual branches taken, this schedule may not be the minimum in the sense of relative scheduling described earlier, however, it reduces the number of $\mathcal{ND}$ operations, thus making it easier to perform the constraint analysis. Also, the cost of implementing control for a bilogic relative scheduler lies somewhere between the control costs for static and relative schedulers.

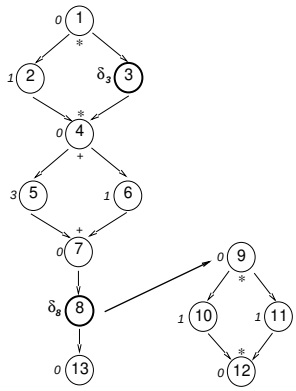
A **bilogic relative schedule** treats an operation offset as a *vector* $\underline{\theta}_{v_j}(v_i)$ representing the (finite) set of possible delays. A bilogic schedule, Ω_{br} then computes the offset vectors such that

$$t(v_i) \geq \max_{a \in \mathcal{A}_b(v_i)} [t_a + \delta(a) + |\underline{\theta}_a(v_i)|_\infty] \tag{4.25}$$

where $|\cdot|_\infty$ refers to the largest element (or the infinity norm) of the vector. The bilogic

anchor set is defined as $\mathcal{A}_b(v_i) = \{v_j \in V : v_j >^* v_i,\ v_j \text{ is } \mathcal{ND}\}$. Once again, the inequality 4.25 can be derived from Eq 4.22 for bilogic flow graphs. Thus a solution to Eq. 4.25 will also satisfy Eq. 4.22 provided $|\underline{\theta}_a(v_i)|_\infty \geq \ell_M(a, v_i)$. The following shows an example of unilogic and bilogic relative schedules.

Example 4.1.1. Unilogic versus bilogic relative schedule for process `example` in Example 3.1.2. The flow graph of the process model is reproduce below. The numbers outside the circle indicate operation delays in cycles.



The assignment of offsets using a relative scheduler and a bilogic relative scheduler are shown below:

Vertex	*Relative offset, θ*				*Bilogic relative offset, $\underline{\theta}$*		
	v_1	v_3	v_4	v_8	v_1	v_3	v_8
1	-	-	-	-	-	-	-
2	0	-	-	-	0	-	-
3	0	-	-	-	0	-	-
4	1	0	-	-	1	0	-
5	-	-	0	-	1	0	-
6	-	-	0	-	1	0	-
7	1	0	0	-	(2,4)	(1,3)	-
8	1	0	0	-	(2,4)	(1,3)	-
9	1	0	0	-	(2,4)	(1,3)	-
10	-	-	-	0	-	-	0
11	-	-	-	0	-	-	0
12	-	-	-	1	-	-	1
13	1	0	0	0	(2,4)	(1,3)	0

where a '-' indicates that the start time of the operation is not affected by the particular anchor vertex. According to the **relative schedule** (Inequality 4.24), the

start time of a vertex v_7, for example, is given by $t(v_7) = \max\{t(v_1) + 1, t(v_3) + \delta_3, t(v_4) + \delta_4\}$. Note that $\delta(v_1) = 0$. For a **bilogic relative scheduler**, the vertex v_3 is no longer an anchor but a variable delay vertex. The offsets are now computed as vectors of possible delay values, as shown below: Thus the start time for vertex, v_7 in this case is given by $t(v_7) = \max\{t(v_1) + 4, t(v_3) + \delta_3 + 3\}$. □

From Section 3.3.4 we recall that for a given flow graph model, $G = (V, E)$, an **implementation**, $\mathcal{I}(G)$ of G refers to the selection of a delay function, δ, that assigns execution delay to simple non-$\mathcal{ND}$ vertices in $V(G)$ and to the choice of a runtime scheduler, Υ.

Definition 4.1 *Given an implementation, $\mathcal{I}(G)$, of a flow graph model, G, a constraint is considered* **satisfiable** *if there exists a solution to the corresponding constraint inequality (Eqs. 3.16, 3.17, 3.18, 3.19) that also satisfies the basic scheduling Inequality 4.22.*

A particular assignment of start times to operations is referred to as a schedule of the operations. For constraint analysis purposes, it is not necessary to determine a schedule of operations, but only to verify the *existence* of a schedule. Since there can be many possible schedules, constraint satisfiability analysis proceeds by identifying conditions under which no solutions are possible.

A timing constraint is considered *inconsistent* if it can not be satisfied by *any* implementation of the flow graph model. A set of timing constraints is considered *mutually inconsistent* if these constraints can not be satisfied by any implementation of the flow graph model. Since the consistency of constraints is independent of the implementation, these are related to the structure of the graphs.

Timing constraint analysis is performed in stages and in order of increasing non-determinism in the model. We first consider the satisfiability of min/max delay constraints followed by the execution rate constraints. The questions about constraint satisfiability are answered in the context of the scheduling schemes discussed in this section. The emphasis in satisfiability analysis is in the determination of constraint satisfiability without relying on runtime (or data-dependent) information. We identify the cases where such a (deterministic) analysis fails and develop the bounds on operation delays in order to satisfy imposed constraints. A notion of marginal satisfiability is developed that relates the likelihood of constraint violation to the probability of violation of delay bounds.

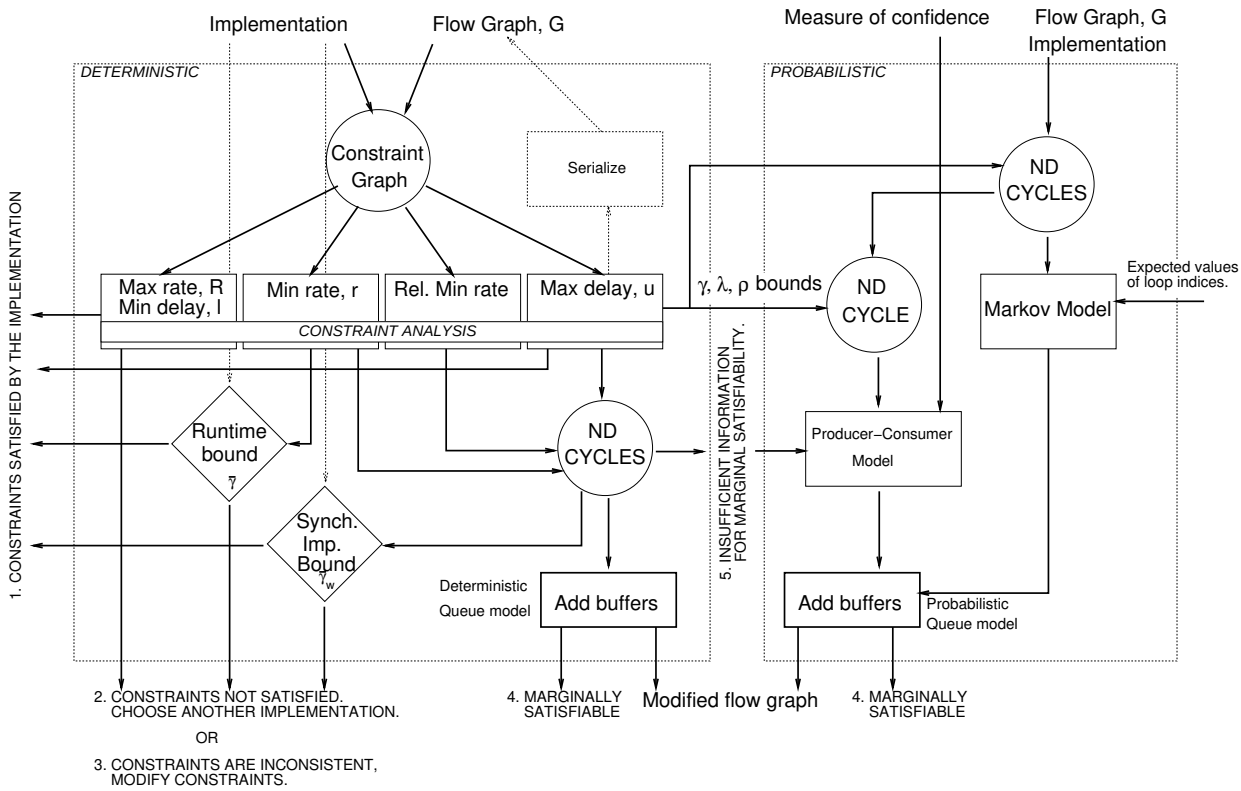


Figure 24: *General flow of constraint analysis.*

Figure 24 shows the flow of the constraint analysis. The given flow graph model with an implementation and a set of min/max delay and execution rate constraints is input to deterministic constraint analysis that relies on a constraint graph model to determine if the constraints are satisfiable. If the constraints are satisfiable (answer 1 in Figure 24), then the choice of hardware or software implementation is acceptable and the constraint analysis is complete. This means that there exists a possible detailed implementation of the graph model in hardware or software for which the constraints can be satisfied. Conversely, a given set of constraints may be violated by an implementation (answer 2), for example, the operation delays may not be fast enough for the choice of hardware or software. If the constraints are not satisfiable by either hardware or software implementations, there is a possibility that constraints may be inconsistent (answer 3). Constraint analysis in all these cases is complete. On the other hand, constraint analysis may be

inconclusive, implying the need for alterations in the *style* of implementation. For example, alternative implementations of the *wait* operation can be explored or buffering can be used to meet execution rate constraints. Such cases are identified by cycles with $\mathcal{ND}$ operations in the constraint graph model, described in the next section.

In presence of cycles with $\mathcal{ND}$ operations in the constraint graph model, constraints may be treated as marginally satisfiable if certain bounds on delay of $\mathcal{ND}$ operations are observed. These (positive) bounds are developed from available slack assuming that the constraints are satisfied (answer 4). In the case of marginally satisfiable constraints, alternative implementations of $\mathcal{ND}$ operations can be explored that improve these bounds. In the last case (answer 5), we need additional information about a measure of confidence (for example, acceptable probability of error) in order to carry out probabilistic analysis.

4.2 Deterministic Analysis of Min/max Delay Constraints

The timing constraints are abstracted in a constraint graph model which is based on the flow graph model. An edge from vertex v_i to v_j with weight, δ, implies that $t_k(v_j) \geq t_k(v_i) + \delta$ for all $k > 0$. This represents a minimum delay constraint on the interval from initiation of v_i to initiation of v_j. A maximum delay constraint from v_i to v_j implies $t_k(v_j) \leq t_k(v_i) + \delta$ which can be rewritten as $t_k(v_i) \geq t_k(v_j) - \delta$ and is indicated as an edge from v_j to v_i with weight $-\delta$. Therefore, maximum delay constraints are represented by edges with negative weights. Since an edge in the (acyclic) flow graph model represents a minimum delay constraint, edges with negative weight are considered backward edges.

Definition 4.2 *The* **timing constraint graph model**, *G_T is defined as $G_T = (V, E, \Delta)$ where the set of edges consists of* forward *and* backward *edges, $E = E_f \cup E_b$ and $\delta_{ij} \in \Delta$ defines the weights on edges such that $t_k(v_i) + \delta_{ij} \leq t_k(v_j)$ for all $k > 0$.*

The constraint graph here does not make any distinction between conjoined and disjoined forks/merge operation nodes. In other words, all forks/merge are considered conjoined. This interpretation is consistent with the constraint Inequality 4.22 where the inequality is defined over all dependencies. It is important to note that the scheduling of

operations for a given set of resources is not affected by the presence of conditionals, since start times for operations must observe worst case path delays. Conditionals do reduce the resource requirements by allowing sharing of resources across mutually exclusive operations [Cam90] which is not an issue here since we assume that the concurrency in the flow graph model can indeed by supported by available resources.

Example 4.2.2. Constraint graph model.

```
#define Const 100
process convert(a,b,c)
        in   port a[3];
        in   port b[2];
        out port c[8];
{
        static sum[8], count[8];
        boolean sample[3], t[1], scale[2], value[8];
        tag tag_a, tag_c;

tag_a:  sample = read(a);
         t  = (sum+sample >= Const);
         if (t) {
                scale = read(b);
                value = sum * scale;
                sum = 0;
                count = 0;
         } else {
                 sum = sum + sample;
                  count++;
         }
tag_c:   write c = count;
          write c = value;
           constraint maxtime from tag_a to tag_c = 12 cycles;
}
```

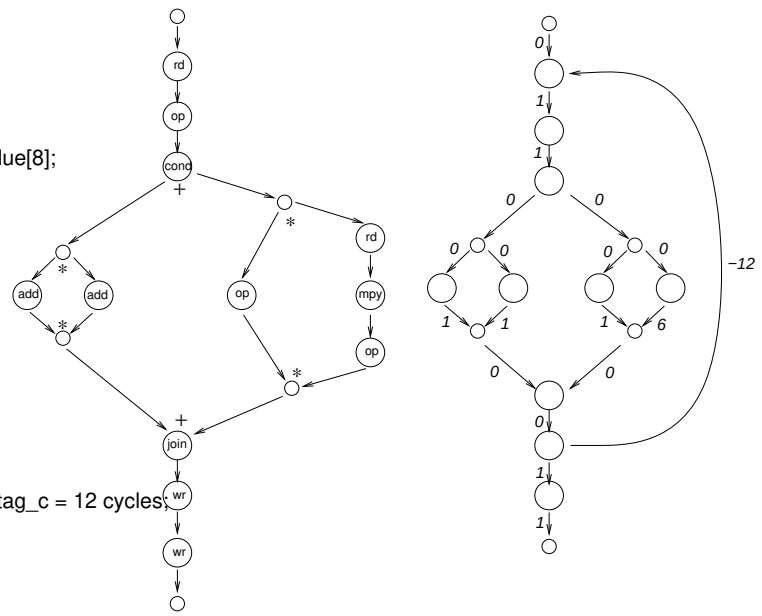


Figure 25: *Constraint Graph Model*

Figure 25 shows the (well-formed) flow graph model and the corresponding constraint graph model for the process `convert`. □

We now examine the use of the constraint graph model in answering question about constraint satisfiability. The following theorem defines the conditions for constraint satisfiability. This theorem occurs in various forms in different application areas. Its proof can be found in, for example, in [CK86] [KM90c] [LW83].

Theorem 4.1 (Static scheduling) *In the absence of any $\mathcal{ND}$ operations, a set of min/max delay constraints is satisfiable if and only if there exist no positive cycles in G_T.*

In the presence of $\mathcal{ND}$ operations, satisfiability analysis attempts to determine the existence of a schedule of operations for all possible (and conceivably infinite) values of the delay of the $\mathcal{ND}$ operations. Clearly, due to variations in operation delays, any static scheduling function that attempts to determine operation initiation times *a priori* must use upper bounds on operation delays such that the initiation times satisfy Inequality 4.22. However, such a schedule may not be minimum. In order to obtain minimum schedules, the operation scheduler can not be static and must have the flexibility to schedule operations *dynamically* based on actual operation delays. As explained above, relative and bilogic relative schedulers allow this flexibility. Therefore, constraint satisfiability is checked only for these schedulers.

Using a relative scheduler, a minimum delay constraint is always satisfiable since from any solution that satisfies Inequality 4.24 or 4.25 a solution can be constructed such that $\theta_{v_j}(v_i) \geq \max(\ell(v_j, v_i), l_{ji})$ for each constraint l_{ji}. This solution satisfies both Inequalities 4.22 and 3.16. On the contrary, a maximum delay constraint may not always be satisfiable. A constraint graph is considered *feasible* if it contains no positive cycle when the delay of $\mathcal{ND}$ operations is assigned to zero.

The following theorem due to Ku and De Micheli [KM92b] lays out a necessary and sufficient condition for to determine the satisfiability of constraints in presence of $\mathcal{ND}$ operations.

Theorem 4.2 (Relative scheduling) *Min/max delay constraints are satisfiable if and only if the constraint graph is feasible and there exist no cycles with $\mathcal{ND}$ operations.*

4.3 Deterministic Analysis of Execution Rate Constraints

Execution rate constraints are constraints on the time interval between invocations of the same operation. In general, this interval can be affected by pipelining techniques since pipelining allows one to initiate an operation sooner than what the total latency of the graph model will allow.

We consider here only non-pipelined implementations of the flow graph models. Limited pipelining of operations is considered in the context of $\mathcal{ND}$-cycles discussed in the following section. Therefore, operations in the graph model are enabled for next

iteration only after completion of the previous iteration:

$$t_{k-1}(v_0) \leq t_{k-1}(v_N) \leq t_k(v_0) \leq t_k(v_N) \qquad \forall\ k > 0. \tag{4.26}$$

where v_0 and v_N refer to the source and the sink vertices respectively.

Consider an I/O operation $v_i \in V(G)$ with data-rate constraints, r_i and R_i. The rate constraints imply

$$\frac{\tau}{R_i} \leq t_k(v_i) - t_{k-1}(v_i) \leq \frac{\tau}{r_i} \qquad \forall\ k > 0. \tag{4.27}$$

τ refers to the cycle time of the clock associated with G. Inequality 4.27 is satisfied if and only if

$$\min_k(t_k(v_i) - t_{k-1}(v_i)) \geq \frac{\tau}{R_i} \qquad \text{[lower bound]} \tag{4.28}$$

$$\max_k(t_k(v_i) - t_{k-1}(v_i)) \leq \frac{\tau}{r_i} \qquad \text{[upper bound]} \tag{4.29}$$

Thus, satisfiability for execution rate constraints is determined by checking for the minimum and maximum delay between any two consecutive invocations of constrained operation. This interval can be expressed as shown in Figure 26, namely:

$$\begin{aligned} t_k(v_i) - t_{k-1}(v_i) &= [t_k(v_i) - t_k(v_0)] + [t_k(v_0) - t_{k-1}(v_N)] + \\ &\quad [t_{k-1}(v_N) - t_{k-1}(v_0)] + [t_{k-1}(v_0) - t_{k-1}(v_i)] \\ &= \lambda_k(v_i) + \gamma_{k-1}(G) + \lambda_{k-1}(G) - \lambda_{k-1}(v_i) \end{aligned} \tag{4.30}$$

where $\lambda_k(v_i)$ refers to execution delay from source vertex v_0 to v_i for the kth execution. $\gamma_{k-1}(G)$ is the delay in rescheduling a graph, that is, the time from completion of $(k-1)^{th}$ execution of G to initiation of the kth execution. From Inequalities 4.22 and 4.26 each of the four components in Inequality 4.30 are non-negative quantities.

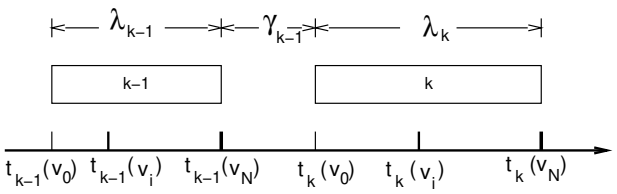


Figure 26: *Operation invocation interval.*

Example 4.3.3. Interval between successive executions of an operation.

Consider the execution of operation 'B' in graph model shown in Example 3.3.6. Its simulation is shown in Figure 17 in Chapter 3. The interval between 1st and 2nd executions of 'B' is given as

$$\begin{aligned} t_2(B) - t_1(B) = 44 - 10 &= \lambda_2(B) + \gamma_1(G_2) + \lambda_1(G_2) - \lambda_1(B) \\ &= 2 + 11 + (11 + 3 \times 4) - 2 = 34 \end{aligned}$$

Similarly, interval between 2nd and 3rd executions of 'B' is expressed as

$$\begin{aligned} t_3(B) - t_2(B) = 54 - 44 &= \lambda_3(B) + \gamma_2(G_2) + \lambda_2(G_2) - \lambda_1(B) \\ &= 2 + 0 + (6 + 1 \times 4) - 2 = 10 \end{aligned}$$

Note that in $\lambda_k(B) = 1$ for all $k > 0$. Note also that the invocation overhead of G_2 is 11 cycles for γ_1 and 0 cycles for γ_2. □

Let us now consider the lower and upper bounds on this interval. These bounds are developed based on the analysis of paths in the flow graph. It follows from Inequality 4.22, that for vertices in a path, $p = \{v_i,\ v_{i+1},\ \cdots, v_j\}$ the following is true for all $k > 0$

$$t_k(v_i) \leq t_k(v_{i+1}) \leq \cdots \leq t_k(v_j) \tag{4.31}$$

It is important to note that even though the actual interval between successive executions is summed as shown in Eq. 4.30, the bounds on this interval can be developed based on analysis of the graph model itself. This is because, in a non-pipelined implementation of G the consecutive execution of an operation corresponds to traversal of a path from source to sink vertex in G. Consider $(k-1)^{th}$ and k^{th} executions of an operation v_i in $V(G)$ as shown in Figure 27. Let $q_{k-1} = \{v_i,\ \cdots,\ v_N\}$ represent the path traversed from v_i to v_N in $k-1^{th}$ execution of G and let $p_k = \{v_0,\ \cdots,\ v_i\}$ be the path traversed from v_0 to v_i in k^{th} execution of G. Using Inq. 4.31 it can be easily shown that $p_k \cup q_{k-1}$ is a path from source to sink in G.

Theorem 4.3 (Maximum rate constraint) *A max-rate constraint, R_i, in G is satisfied if $\ell_m(G) \geq R_i^{-1}$.*

Proof: In order to obtain a lower bound on the interval between two consecutive executions of operation, v_i, we consider the case when the

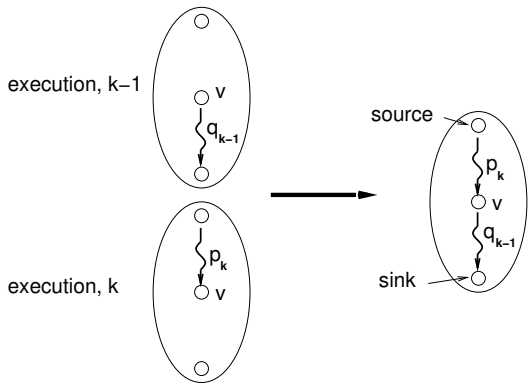


Figure 27: *Consecutive executions of an operation corresponds to traversal of a path in* G.

execution of the graph model is restarted immediately after the completion of the previous execution, that is, $\gamma_{k-1}(G) = 0$. From the discussion above, there exists a path in G that corresponds to consecutive execution of operation v_i. In other words, the interval $t_k(v_i) - t_{k-1}(v_i)$ is bounded by the latency of the graph. Recall that the length vector provides a lower bound on latency of G. The result follows. ♯

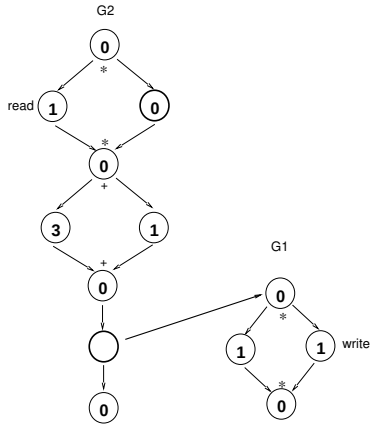
Note that similar to a minimum delay constraints, a maximum rate constraint is always satisfiable. When $\ell_m(G) < R_i^{-1}$ the maximum rate constraint, R_i, can still be satisfied by an appropriate choice of overhead delay that is applied to every execution of G.

Example 4.3.4. Maximum rate constraints.

For the process graph model `convert` shown in Figure 25, $\underline{\ell}(G) = (5, 10)$. Therefore, for non-pipelined implementations of G, the lower bound on the interval between successive initiations of any operation in G is 5 cycles. In other words, the maximum rate for any operation in G is $\frac{1}{5} = 0.2$ cycle^{-1}. For any maximum rate constraint less than 0.2 cycle^{-1} transformation of the flow graph would be needed to ensure satisfaction of the maximum rate constraint.

For the process graph model `example` shown in Figure 17 (reproduced blow) the maximum rate of the `write` operation, determined by $\ell(G_1)$, is 1 cycle^{-1}, whereas the maximum rate of the `read` operation, determined by $\underline{\ell}(G_2) = ((1 \otimes 0) \odot (1 \oplus 3)) \odot (1 \otimes 1) = (3, 5)$ is $\frac{1}{3}$ cycle^{-1}. Any maximum rate constraint larger than or equal to $\frac{1}{3}$ is satisfied by the graph model.

Note that this lower bound, ℓ_m, used for checking the satisfaction of maximum rate constraints, also defines the fastest rate at which an operation in the graph model



can be executed by a non-pipelined implementation. Thus points to the necessary condition for meeting a minimum rate constraint. Sufficient conditions for minimum rate constraints are considered in next section. □

Note that in case of a pipelined implementation of G, the operation v_i can be restarted without waiting for completion of all operations in G. An extreme example of this would be a buffer at an input operation (equivalent to a pipestage containing only v_i), in which case the operation can be enabled after every execution delay of v_i (until the buffer is full).

Upper bound

While the lower bound on time-interval between successive executions of an operation can be derived by analyzing G^*, that is the graph to which the operation belongs and all the graphs *below* in the control-flow hierarchy, the determination of upper-bound on the inter-iteration interval of an operation, requires also estimations of the delays due to operations and graphs that lie *above* the operation in the control-flow hierarchy. In particular, the effect of the runtime scheduler must also be taken into account.

We use following notation to help express the propagation of constraints over the graph hierarchy. For a given graph G, G_+ denotes the parent body that calls the graph G. For a graph, G, G_o refers to the *parent process graph*, that is, the graph at the root of the hierarchy corresponding to a process model.

Note that (static) determination of interval of successive executions of an operation

that is conditionally invoked is undecidable. That is, there may not exist an upper bound on the invocation interval. For example, consider a statement

```
if (condition)
        value = read (a);
```

There is not enough information to determine the rate of execution of the 'read' operation. In order determine constraint satisfiability we need additional input on how frequently the condition is true. For deterministic analysis purposes, we take a two step approach to answering constraint satisfiability:

1. Answer about implementation satisfiability *assuming that the condition is always true*. In other words, the only uncertainty is conditional invocation of the graph which may correspond to the body of a process or a loop operation. This is consistent with the interpretation that a timing constraint specifies a bound on the interval between operation executions, but does not imply *per se* that the operation must be executed.

   Under this assumption, the loops are executed at least once (that is, loops are of the type 'repeat-until') since the 'while' loops are expressed as a conditional followed by a repeat-until loop as explained in Chapter 3.

2. Next we use the rate constraint on the 'read' operation as the additional information about frequency of invocation of the condition. That is, the rate constraint serves as a *property* of the environment in continuing the rate constraint analysis. This way, constraints are source of additional input which is far more convenient to specify than probabilities of conditions taken. An alternative approach would be to use simulations to collect data on the likelihood of the condition being true and use it to derive constraint satisfiability.

The actual execution delay or the latency, $\lambda(G)$, refers to the delay of the longest path in G. This path may contain $\mathcal{ND}$ operations in which case the latency can not be bounded. We examine the two cases separately.

Case I: G contains no $\mathcal{ND}$ operations. The latency of G takes one of the finite values given by $\underline{\ell}(G)$. Equations 3.4 through 3.9 define the formulae for calculation of $\underline{\ell}$. An upper bound on the operation interval is then given by:

$$\begin{aligned} \max_k(t_k(v_i) - t_{k-1}(v_i)) &\leq \max_k [\gamma_{k-1}(G) + \lambda_{k-1}(G)] \\ &\leq \max_k \gamma_k(G) + \ell_M(G) \end{aligned} \tag{4.32}$$

Let us now examine the delay $\gamma_k(G)$. The overhead $\gamma_k(G)$ represents the delay $[t_{k+1}(v_0(G)) - t_k(v_N(G))]$ and can be thought of as an additional delay operation in series with the sink operation, $v_N(G)$. If G is not a root-level flow graph, then there exists a parent flow graph G_+ that calls G by means of a link operation, say v. The upper bound on this interval is derived when the kth and $(k+1)^{th}$ invocations of G correspond to separate invocations of the link operation $v \in V(G_+)$. That is,

$$\begin{aligned} \gamma_k(G) &= t_{k+1}(v_0(G)) - t_k(v_N(G)) \\ &\leq t_{j+1}(v) - t_j(v) - x_j \cdot \lambda_k(G) \\ &\leq \max_j[t_{j+1}(v) - t_j(v)] - \min_j x_j \cdot \min_k[\lambda_k(G)] \end{aligned} \tag{4.33}$$

where x_j is the number of times the flow graph G is invoked for the jth execution of operation v. By definition, G is invoked at least once for each execution of v, i.e., $\min_j x_j = 1$. Therefore, from Inq. 4.32 and 4.33,

$$\gamma_k(G) \leq \overline{\gamma}(G) \doteq [\ell_M(G_+) + \overline{\gamma}(G_+)] - \ell_m(G) \tag{4.34}$$

Note that by definition, $\ell_M(G_+) \geq \ell_M(G) \geq \ell_m(G)$, therefore, $\overline{\gamma}$ is always a positive quantity.

Lemma 4.1 (Minimum rate constraint with no $\mathcal{ND}$) *A minimum rate constraint on an operation $v_i \in V(G)$, where G contains no $\mathcal{ND}$ operations is satisfiable if*

$$\overline{\gamma}(G) + \ell_M(G) \leq \frac{\tau}{r_i} \tag{4.35}$$

where the overhead term $\overline{\gamma}(G)$ is defined by Equation 4.34.

Proof: Follows from Inq. 4.29, 4.32 and 4.34. ♯

Clearly, a bound on the overhead delay $\gamma_k(G)$ implies existence of a bound on the invocation interval of G_+, and by induction, bound on the invocation interval of all graphs in the parent hierarchy. In particular, the bound on the invocation interval of the parent process graph, G_o, corresponds to the bound on the delay due to the runtime scheduler overhead. This places restrictions on the choice of the runtime scheduler such that a bound on the scheduling interval can indeed be placed (see Section 5.2). Note that a bound on $\gamma_k(G)$ does not necessarily imply a bound on the latency of G. This is illustrated by Example 4.3.5 below. An immediate consequence of the above (sufficient) condition for satisfiability of minimum rate constraint is that question about the constraint satisfiability can be *propagated* as a minimum rate constraint on the link operation in the parent graph model. The following lemma defines this concept more precisely.

If a given implementation of a flow graph G satisfies a minimum rate constraint r_i on an operation G, we say that G satisfies the rate constraint, r_i.

Lemma 4.2 (Constraint propagation) *A flow graph G satisfies a minimum rate constraint r_i if its parent graph G_+ satisfies a minimum rate constraint $\left[\frac{\tau}{r_i} - \Delta\ell(G)\right]^{-1}$, where $\Delta\ell(G) \doteq \ell_M(G) - \ell_m(G)$.*

Proof: If G_+ satisfies a minimum rate constraint $\left[\frac{\tau}{r_i} - \Delta\ell(G)\right]^{-1}$ then from Lemma 4.1,

$$\begin{aligned} \overline{\gamma}(G_+) + \ell_M(G_+) &\leq \frac{\tau}{r_i} - \Delta\ell(G) \\ \Rightarrow \overline{\gamma}(G) + \ell_m(G) &\leq \frac{\tau}{r_i} - (\ell_M(G) - \ell_m(G)) \qquad [4.34] \\ \Rightarrow \overline{\gamma}(G) + \ell_M(G) &\leq \frac{\tau}{r_i} \end{aligned}$$

$\Rightarrow G$ satisfies minimum rate constraint r_i. ♯

In order to obtain a bound on the runtime scheduler overhead, Equation 4.34 can be unrolled until the parent graph corresponds to the (unconditionally invoked) process model, G_o for which $\gamma(G_o) = \gamma_o$. Thus,

$$\overline{\gamma}(G) = \sum_{G_i = G_+}^{G_o} \Delta\ell(G_i) + \overline{\gamma}_o + [\ell_m(G_o) - \ell_m(G)] \qquad (4.36)$$

where $\overline{\gamma}_o = \overline{\gamma}(G_o)$ is the bound on the delay due to the runtime scheduler.

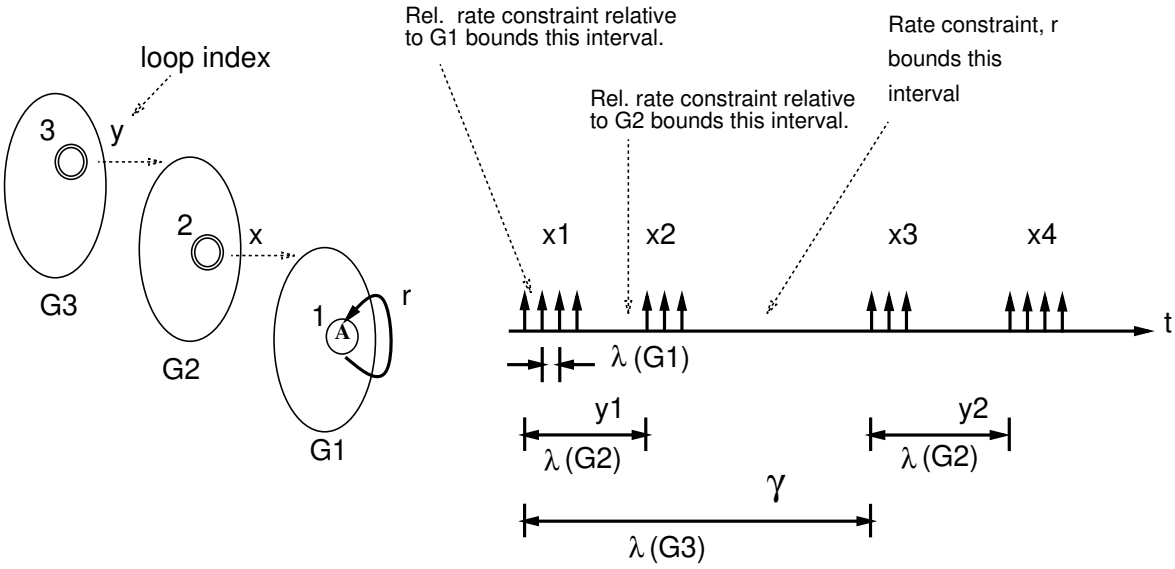


Figure 28: *Upward propagation of minimum execution rate.*

Example 4.3.5. Minimum rate propagation.

Figure 28 shows the constraint model corresponding to the graph models in Example 3.3.6. Recall

$$\begin{array}{lll} \underline{\ell}(G_1) = (3,4) & \Delta\ell(G_1) = 4-3 & = 1 \\ \underline{\ell}(G_2) = (9,10,14,15) & \Delta\ell(G_2) = 15-9 & = 6 \\ \underline{\ell}(G_3) = (13,14,18,19,20,21,23,24,25,26) & \Delta\ell(G_3) = 26-13 & = 13 \end{array}$$

First we show the intuition behind rate constraint satisfiability, followed by the use of constraint propagation to achieve the same result.

A minimum rate constraint is specified on operation 'A' in G_1 that constitutes loop body of operation 2 in G_2 with loop index, x, which in turn is a loop body of operation 3 in G_3. Let $r_A = 1/100$, $r_A^{G_1} = 1/5$, $r_A^{G_2} = 1/25$ and $r_A^{G_3} = 1/50$ cycle^{-1}. Recall, that r_A^G refers to a minimum rate constraint *relative to* G.

Let us first consider, $r_A^{G_1} = 1/5$ cycle^{-1}. Since this constraint is relative to G_1, therefore, there is no overhead in invocation of G_1, i.e., $\overline{\gamma}(G_1) = 0$. Since

$$[\overline{\gamma}(G_1)] + \ell_M(G_1) = 4 \leq 1/\frac{1}{5} = 5$$

Therefore, the constraint $r_A^{G_1} = 1/5$ is satisfied. Similarly, constraint $r_A^{G_2} = 1/25$ is satisfied since

$$\begin{array}{rcl} \overline{\gamma}(G_1) + \ell_M(G_1) & = & [\ell_M(G_2) + \overline{\gamma}(G_2) - \ell_m(G_1)] + \ell_M(G_1) \\ & = & [15 + 0 - 3] + 4 = 16 \leq 1/\frac{1}{25} = 25 \end{array}$$

Constraint $r_A^{G_3} = 1/50$ is satisfied since

$$\begin{aligned}\overline{\gamma}(G_1) + \ell_M(G_1) &= [\Delta\ell(G_3) + \Delta\ell(G_2) + \overline{\gamma}(G_3) + \ell_m(G_3) - \ell_m(G_1)] + \ell_M(G_1) \\ &= [13 + 6 + 0 + 13 - 3] + 4 = 33 \leq 1/\frac{1}{50} = 50\end{aligned}$$

Finally, for the minimum rate constraint $r_A = 1/100$ we should also consider the overhead $\overline{\gamma}_o$ due to the runtime scheduler which adds to the bound of 33 cycles on successive intervals of operation 'A' *relative to* G_3. Therefore, a r_A is satisfied if the delay due to the runtime scheduler is less than or equal to $100 - 33 = 67$ cycles.

Alternatively, $r_A = 1/100$ can be propagated as a rate constraint of $\frac{1}{100-1} = 1/99$ on G_2 which is in turn propagated as a rate constraint of $\frac{1}{99-6} = 1/93$ on G_3. This constraint on G_3 is satisfied for a bound of $93 - \ell_M(G_3) = 93 - 26 = 67$ cycles on the delay due to the runtime scheduler. □

Theorem 4.4 (Minimum rate constraint with no $\mathcal{ND}$) *A minimum rate constraint on operation, $v_i \in V(G)$, where G contains no $\mathcal{ND}$ operations is satisfiable if the minimum available overhead for the runtime scheduler, γ_{avail} is greater than the maximum delay $\overline{\gamma}_M$ offered by the chosen runtime scheduler. That is,*

$$\overline{\gamma}_{\text{avail}} \doteq \frac{\tau}{r_i} - \ell_M(G) - \sum_{G_i=G_+}^{G_o} [\ell_M(G_i) - \ell_m(G_i)] - [\ell_m(G_o) - \ell_m(G)] \geq \overline{\gamma}_M \quad (4.37)$$

Proof: The maximum delay $\overline{\gamma}_M$ due to the runtime scheduler defines the overhead $\overline{\gamma}_o$ of the process model G_o.

$$\begin{aligned}
& \overline{\gamma}_{\text{avail}} \doteq \tfrac{\tau}{r_i} - \ell_M(G) - \textstyle\sum_{G_i=G_+}^{G_o} [\ell_M(G_i) - \ell_m(G_i)] - [\ell_m(G_o) - \ell_m(G)] && \geq \overline{\gamma}_M \\
\Rightarrow\ & \tfrac{\tau}{r_i} \geq \ell_M(G) + \textstyle\sum_{G_i=G_+}^{G_o} \Delta\ell(G_i) + [\ell_m(G_o) - \ell_m(G)] + \overline{\gamma}_M \\
\Rightarrow\ & \tfrac{\tau}{r_i} \geq \ell_M(G) + \left\{\textstyle\sum_{G_i=G_+}^{G_o} \Delta\ell(G_i) + [\ell_m(G_o) - \ell_m(G)] + \overline{\gamma}_o\right\} \\
\Rightarrow\ & \tfrac{\tau}{r_i} \geq \ell_M(G) + \overline{\gamma}(G) && [4.36] \\
\Rightarrow\ & r_i \text{ is satisfied.} && [\text{Lemma } 4.1]
\end{aligned}$$

♯

In summary, a minimum execution rate constraint on a graph model, G, that contains no $\mathcal{ND}$ operations is translated as an upper bound, $\overline{\gamma}$, on the delay of the runtime system which checked by comparing it against $\overline{\gamma}_M$. Note that if the graph G is not a root level graph, then there exists a parent graph G_+ with a link operation that calls G. However, the

unbounded delay due to this $\mathcal{ND}$ operation does not affect satisfiability of the minimum rate constraints on operations in G as shown by the example above. In general, the delay of an $\mathcal{ND}$ operation affects satisfiability of a minimum rate constraint applied on an operation other than the operations linked with the $\mathcal{ND}$ operation. This is included in the case considered next.

Case II: G contains $\mathcal{ND}$ operations. In presence of $\mathcal{ND}$ operations in G, the latency, $\lambda(G)$ can no longer be bounded by the longest path length, $\underline{\ell}$, in G. In addition, if G is not a root-level flow graph, its the overhead $\gamma(G)$ may also not be bounded by the maximum path length of its parent graph. For the sake of simplicity, let us first consider the (relative) minimum rate constraint on a graph model with zero overhead, that is, $\gamma_k(G) = 0$ for all $k > 0$. Such a rate constraint then bounds the latency of the graph model and is represented as a backward edge (that is, a maximum delay constraint) from the sink vertex to the source vertex in the constraint graph model of G. Since G is a connected graph, such a constraint invariably leads to a $\mathcal{ND}$-cycle in the constraint graph. According to Theorem 4.2, the maximum delay constraint can be satisfied only by bounding the delay of the $\mathcal{ND}$ operation, that is, by transforming the $\mathcal{ND}$ operation into a non-$\mathcal{ND}$ operation.

Since $\mathcal{ND}$ operations represent synchronization or data-dependent loop delay, the implications of developing bounds on the delay of these operations must be carefully analyzed. While a discussion on this subject appears in Section 4.5, here we briefly capture the motivation for developing the bounds.

- Let us first consider synchronization related $\mathcal{ND}$ operations. Since there are multiple ways of implementing a synchronization operation, the effect of the bound is to choose those implementations which are *most likely* to satisfy the minimum rate constraint. Thus, a bound on the delay of the synchronization refers to a bound on the delay offered by the *implementation* of the $\mathcal{ND}$ operation. The implementation delay of a synchronization operation is referred to as the *synchronization overhead*, γ_w. Due to the availability of multiple concurrent execution streams in hardware, this overhead is zero. For software, γ_w, delay is determined by the implementation

of the wait operation by the runtime scheduler. For example, a common implementation technique is to force a *context switch* in case an executing program enters a wait state. Here, γ_w would be twice the context-switch delay to account for the round-trip delay. For such an implementation, the minimum rate constraint is interpreted as the rate supportable by an implementation. With this interpretation, the $\mathcal{NCD}$ operations are considered non-$\mathcal{NCD}$ operations with a fixed delay, γ_w.

- Next, the data-dependent loop operations use a data-dependent *loop index* that determines the number of times the loop body is invoked for each invocation of the loop operation. The delay offered by the loop operation is its loop index times the latency of the loop body. As mentioned earlier, at the leaf-level of graph hierarchy, the latency of the loop body is given by its path length vector. The elements of a path length vector consists of lengths of all paths from source to sink and these are bounded. In the case the constrained graph model contains at most one loop operation, v, on a path from source to sink, the minimum rate constraint can be seen as a bound on the number of times the loop body G_v corresponding to the loop operation, v, is invoked. This bound on loop index, x, is given by Equation 4.39 that is derived later. This bound $\overline{x}$ is then treated as a *property* of the loop operation, consequently making it a non-$\mathcal{NCD}$ operation with a bounded delay for carrying out further constraint analysis. Verification of these bounds requires additional input from the user.

  For a relative minimum rate constraint constraint relative to G, the overhead term $\overline{\gamma}(G)$ in Equation 4.39 is assigned zero value. In general, however, the satisfiability of a minimum execution rate constraint also includes a bound on the invocation delay γ of G as per Equation 4.36. Clearly, a bound on $\gamma(G)$ implies a bound on the latency of G_+ which is equivalent to a minimum rate constraint on an operation in G_+. However, this minimum rate constraint does not bound the loop index of link operation associated with G. The constraint satisfiability is then continued until G_+ corresponds to a process body, G_o.

Presence of multiple $\mathcal{NCD}$ operations in G and G_+ present a more difficult case since a minimum rate bounds the effective delay which is now a function of multiple loop

indices. For example, consider $\mathcal{ND}$ operations v_i and v_j representing loops in the flow graph, that have a (transitive) dependency in the flow graph model (and thus belong to the same path from source to sink vertices). In this case the satisfaction of minimum rate constraint requires that the combined delay due to v_i and v_j be bounded. This requirement can be expressed as:

$$\frac{x_i}{\overline{x}_i} + \frac{x_j}{\overline{x}_j} \leq 1 \tag{4.38}$$

where $\overline{x}_i$ defines the bound on loop index x_i assuming $x_j = 0$ for all $j \neq i$. Note that the above equation also applies in the case the $\mathcal{ND}$ operations v_i and v_j belong to nested flow graphs. The satisfiability of Equation 4.38 is answered in two ways:

Deterministically: by substituting x_i by the upper bound $\widehat{x}_i$ in Equation 4.38. This upper bound is additional input from the user (in practice this can also be determined by the bit-width of the variable used for loop index).

Statistically: by treating x_i as random variables. Then the constraint is satisfied in probability, if Inequality 4.38 is satisfied over expected values of the random variables. These expected values are the additional input from the user needed to check satisfiability.

In both the cases, in presence of multiple $\mathcal{ND}$ operations that lie on the same path from source to sink in a graph model, it is not possible to answer question about constraint satisfiability without additional input from the environment with which the system interacts. We consider this problem further in Section 4.7.

Theorem 4.5 (Minimum rate constraint with $\mathcal{ND}$) *Consider a flow graph G with an $\mathcal{ND}$ operation v representing a loop in the flow graph. A minimum rate constraint r_i on operation $v_i \in V(G)$ and $v_i \neq v$ is satisfiable if the loop index, x_v indicating the number of times G_v is invoked for each execution of v is less than the bound $\overline{x}_v$ where*

$$\overline{x}_v \doteq \left\lfloor \frac{\tau r_i^{-1} - \overline{\gamma}(G) - \ell_M(G) + \mu(v)}{\ell_M(G_v)} \right\rfloor + 1 \tag{4.39}$$

where $\mu(v)$ refers to the mobility *of operation v and is defined as the difference in length*

of the longest path that goes through v and ℓ_M.[1] *G_v refers to the graph model called by the $\mathcal{ND}$ operation v and the overhead bound, $\overline{\gamma}(G)$ is defined by Equation 4.36.*

Proof: The maximum interval between successive executions of operation $v_i \in V(G)$ is given by the maximum latency of G and its maximum overhead, $\overline{\gamma}(G)$. (See Inequality 4.29 and the following discussion.). The latency of G is can be defined as the maximum over the lengths of all paths from source to sink vertices. Let p_v represent the longest path from source to sink that goes through operation v.

$$\lambda(G) \leq \ell_M(p_v) + (x_v - 1) \cdot \ell_M(G_v)$$

Note that $\ell_M(G)$ is computed by treating all link vertices as call link vertices (see Chapter 3) and, therefore, it includes the delay due to one execution of each loop body, hence the second term in equation above represents the additional component to the latency due to the $(x_v - 1)$ invocations of the loop flow graph G_v.

The length of the longest path from source to sink determines the value of $\ell_M(G)$. The vertex v may or may not lie on the longest path from source to sink operations. This slack between $\ell_M(G)$ and the length of the longest path through v is captured by the mobility $\mu(v)$ of operation v. That is, $\ell_M(p_v) = \ell_M(G) - \mu(v)$.

For satisfiability of constraint r_i, we require that

$$\begin{aligned} & \overline{\gamma} + \max_k \lambda_k(G) \leq \frac{\tau}{r_i} \\ \Rightarrow \quad & \overline{\gamma} + \{\ell_M(G) - \mu(v) + (x_v - 1) \cdot \ell_M(G_v)\} \leq \frac{\tau}{r_i} \\ \Rightarrow \quad & x_v \leq \left\lfloor \frac{\frac{\tau}{r_i} - \overline{\gamma}(G) - \ell_M(G) + \mu(v)}{\ell_M(G_v)} \right\rfloor + 1 \end{aligned}$$

This provides the bound on every loop index in G. In addition, following discussion earlier in this section, if multiple $\mathcal{ND}$ operations lie on the same path from source to sink, Equation 4.38 must also be satisfied. ♯

[1] The mobility is computed in $O(|E(G)|)$ time as the difference in starting times of ALAP and ASAP schedules of a deterministic delay flow graph constructed by considering all link vertices to be call link vertices with delay as the maximum path length of the called graphs.

Remark 4.1 *In presence of multiple $\mathcal{ND}$ operations, the minimum rate constraint on an operation v_i is satisfied if each loop index x_v is bounded as in Equation 4.39 above for all $\mathcal{ND}$ operations $v \in V(G)$ and $v \neq v_i$, and Equation 4.38 is satisfied.*

Example 4.3.6. Bound on loop index due to minimum execution rate constraint.

Consider a minimum rate constraint of 0.02 /cycle on operation 'B' in graph model, G_2 shown in Example 3.3.6. Let the maximum delay due to the runtime scheduler be $\overline{\gamma}_M = 0$ (for example, hardware implementation). The bound on the loop index for operation v_2 is calculated as follows:

$$\begin{aligned}
\overline{\gamma}(G_2) &= \Delta\ell(G_3) + \overline{\gamma}_M + \ell_m(G_3) - \ell_m(G_2) \\
&= 13 + 0 + 13 - 9 = 17 \\
\overline{x}_2 &= \left\lfloor \frac{\tau r_B^{-1} - \overline{\gamma}(G_2) - \ell_M(G_2) + \mu(v_2)}{\ell_M(G_1)} \right\rfloor + 1 \\
&= \left\lfloor \frac{50 - 17 - 15 + 0}{4} \right\rfloor + 1 = 5.
\end{aligned}$$

With this bound on loop index, the $\mathcal{ND}$ operation v_2 has a bound on its delay of 20 cycles.

On the other hand, a *relative* rate constraint, $r_B^{G_2}$ of 0.02 /cycle leads to a bound on loop index of

$$\overline{x}_2 = \left\lfloor \frac{50 - 0 - 15 + 0}{4} \right\rfloor + 1 = 9.$$

with this bound the delay of v_2 is less than 36 cycles. □

In summary, satisfaction of the bounds on delay of $\mathcal{ND}$ operations requires additional information from their implementations (such as context switch delay, possible loop index values) against which the questions about satisfiability of minimum rate constraint can be answered. Because of these bounds, there is now a certain *measure* of constraint satisfiability that approaches certainty as the derived bound approaches infinity. More importantly, having bounds derived from timing constraints makes it possible to seek transformations to the system model which tradeoff these measures of constraint satisfiability against implementation costs. In the next section, we examine conditions under which these bounds can be extended by modifying the structure of the flow graphs with $\mathcal{ND}$ cycles.

4.3.1 Procedure

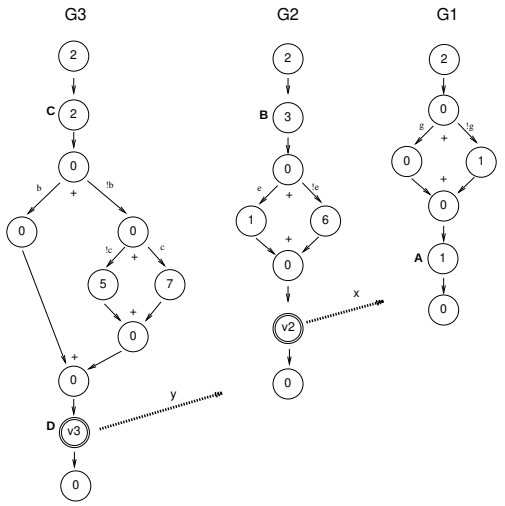
Given a flow graph model G with min/max delay and execution rate constraints, the constraint analysis proceeds bottom-up. The leaf-level flow graphs do not contain any loop $\mathcal{ND}$ operations. For constraint analysis purposes, the graph bodies of procedure calls are considered flattened into the calling graph model.

The following procedure *check_satisfiability* outlines the algorithm. The input is a set of graph models with min/max and rate constraints along with an implementation, $\mathcal{I} = (\Upsilon, \Delta)$. Its output is null if the constraints are satisfiable (answer 1), else either G is unsatisfiable (answers 2 and 3) or it returns bounds on the delay of $\mathcal{ND}$ operations that would make constraints satisfiable (answers 4 and 5). As discussed earlier, the wait operation is replaced by its implementation which is either a fixed delay operation or a loop operation (representing a busy-wait implementation). The constraint analysis proceeds from identification of cycles in the constraint graph, G_T. The cycles are found by considering each backward edge at a time and enumerating all cycles caused by the backward edge. If the length of a cycle is positive, the constraint graph is not feasible and, therefore, constraints can not be satisfied. Recall that for length calculation purposes, the loop link operations are treated same as call link operations. In presence of cycles that contain loop link operations, the algorithm derives bounds on the loop index of each loop link operation. In case of series or nested loop link operations verification of additional constraints on loop indices is done separately.

```
check_satisfiability(G) {
        for v ∈ V(G) {
            if v = loop                          /* recursively go to leaf-level graph */
                check_satisfiability(G_v);
        }
▷I      construct G_T                            /* construct the constraint graph model */
▷II     if (cycle-set = find-cycles(G_T)){       /* check for min/max */
            for Γ ∈ cycle-set {                  /* identify cycles caused by backward edges */
                if (ℓ_M(Γ) > 0)                  /* find positive length cycles */
                    return (G is unsatisfiable);  /* not feasible */
                for v ∈ Γ and v ∈ ND {           /* identify ND cycles */
                    print δ_v = u − ℓ_M(Γ);
                    bound delay of v = δ_v;      /* bound on ND delay using constraints */
                    mark v as non-ND;            /* now treat this delay bound as a property */
                }
            }
```

```
        }
        s = ⌈ℓ_m(G) − τ · max_i R_i^{-1}⌉          /* check for max rate */
        if s ≥ 0
            return (G is satisfied);
        else {                                      /* need to add null operations – */
            add NOP with δ = s;                     /* – to ensure lower bound on delay */
            update ℓ(G_o);                          /* modified flow graph */
            check_satisfiability(G);
        }
        if G_+ exists                               /* check for min rate */
▷III        impose constraint [τ/r_i − Δℓ(G)]^{-1} on link operation in G_+;  /* propagate r_i */
}
```

Example 4.3.7. Procedure *check-satisfiability* on Example 3.3.6. For convenience the graph model is reproduced below.



Let us assume the following imposed constraints:

$r_A = 1/100, r_A^{G_1} = 1/6, r_A^{G_2} = 1/40, r_B = 1/50, r_B^{G_2} = 1/30, r_C = 1/200$
$u_{CD} = 12, R_B = 0.5, \overline{\gamma}_M = 20.$

Recall

$$\begin{array}{lll} \underline{\ell}(G_1) & = (3,4) & \Delta\ell(G_1) = 1 \\ \underline{\ell}(G_2) & = (9,10,14,15) & \Delta\ell(G_2) = 6 \\ \underline{\ell}(G_3) & = (13,14,18,19,20,21,23,24,25,26) & \Delta\ell(G_3) = 13 \end{array}$$

There are three main steps to the constraint analysis procedure: construction of the constraint graph which is done by adding forward edges for minimum delay and

maximum rate constraints, and backward edges for maximum delay and (relative) minimum rate constraints. Identification of cycles by path enumeration for each of the backward edges in the constraint graph and finally the propagation of minimum rate constraints up the graph hierarchy. We show these three steps for this example.

The procedure first considers $\mathbf{G_1}$:

$\triangleright$**I** : In the constraint graph of G_1, there are 3 backward edges with following weights:

$$\begin{aligned}
r_A^{G_1} = 1/6 &\Rightarrow -6 \\
r_A^{G_2} = 1/40 &\Rightarrow -[40 - \gamma(G_1)|_{\gamma(G_2)=0}] \\
&= -[40 - \gamma(G_2) - \ell_M(G_2) + \ell_m(G_1)] \\
&= -[40 - 0 - 15 + 3] \\
&= -28 \\
r_A = 1/100 &\Rightarrow -[100 - \gamma(G_1)] = -(100 - [\gamma(G_2)] - 15 + 3) \\
&= -(88 - [\Delta\ell(G_3) + \overline{\gamma}_M + \ell_m(G_3) - \ell_m(G_2)]) \\
&= -(88 - [13 + 20 + 13 - 9]) \\
&= -51
\end{aligned}$$

$\triangleright$**II** : The maximum forward path length is $4 < 6$
$\Rightarrow$ no positive cycles
$\Rightarrow$ The constraints are feasible. Further, G_{T1} contains no $\mathcal{ND}$ cycles.

$\triangleright$**III** : Propagate minimum rate constraints to G_2 $\Rightarrow$
$$\begin{aligned}
r_A^{G_1} &\Rightarrow \text{ not propagated.} \\
r_A^{G_2} &\Rightarrow r_{v_2}^{G_2} = 1/(28-1) = 1/27 \\
r_A &\Rightarrow r_{v_2} = 1/(51-1) = 1/50
\end{aligned}$$

For $\mathbf{G_2}$:

$\triangleright$**I** : In the constraint graph of G_2, there are 4 backward edges with following weights:

$$\begin{aligned}
r_B = 1/50 &\Rightarrow -(50 - \overline{\gamma}(G_2)) = -(50 - 37) = -13 \\
r_{v_2}^{G_2} = 1/27 &\Rightarrow -27 \\
r_{v_2} = 1/50 &\Rightarrow -50 \\
r_B^{G_2} = 1/30 &\Rightarrow -30
\end{aligned}$$

$\triangleright$**II** : r_B is infeasible since it leads to a positive cycle with weight=15-13=2. Rest are feasible. Next, the constraint graph contains $\mathcal{ND}$ cycles with a single $\mathcal{ND}$ operation v_2 for each of the three (feasible) backward edges. Of these only one, namely $r_B^{G_2}$ bounds the delay due to the $\mathcal{ND}$ operation by the following upper bound on loop index, $\overline{x}(v_2) = \left\lfloor \frac{30-0-15+0}{4} \right\rfloor + 1 = 4$. With this bound the delay of the loop operation, v_2 is bound below 16 cycles.

$\triangleright$**III** : Propagate minimum rate constraints to G_3 $\Rightarrow$
$r_B \Rightarrow$ Infeasible. Not propagated.
$r_{v_2}^{G_2} = 1/27 \Rightarrow$ Not propagated.
$r_{v_2} = 1/50 \Rightarrow r_{v_3} = 1/(50-6) = 1/44$
$r_B^{G_2} = 1/30 \Rightarrow$ Not propagated.

Finally for G_3:

$\triangleright$**I** : In the constraint graph of G_3, there are 3 backward edges with following weights:

$$\begin{aligned} u_{CD} = 12 &\Rightarrow -12 \\ r_C = 1/200 &\Rightarrow -(200 - \overline{\gamma}(G_3)) = -180 \\ r_{v_3} = 1/44 &\Rightarrow -44 \end{aligned}$$

$\triangleright$**II** : There are no positive cycles, so the constraint graph is feasible. Further, two backward edges lead to $\mathcal{ND}$ cycles. Only one of them, r_C constraints the delay of the $\mathcal{ND}$ operation, v_3. The bound on the loop index, $\overline{x}_3 = \left\lfloor \frac{180 - \ell_M(G_3) + \mu(v_3)}{\ell_M(G_2)} \right\rfloor + 1 = 11$. With this bound the delay of v_3 is ≤ 165.

$\triangleright$**III** : There is no parent graph to propagate the minimum rate constraints. □

4.4 Min/max Constraints Across Graph Models

The algorithm presented in the previous section carries out constraint analysis on constraint graphs by considering one flow graph at a time. We now consider timing constraint between operations that belong to two separate flow graphs G_1 and G_2. The satisfiability constraints that span across flow graphs is affected by the relationships between the flow graphs. As shown in Figure 29 there are following three types of relationships between flow graphs:

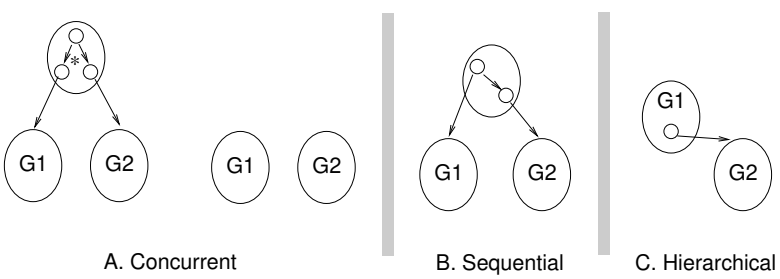


Figure 29: *Relationships between flow graphs*

Case A: G_1 and G_2 are concurrent. This refers to the case when invocation paths to G_1 and G_2 are disjoint. If the graphs G_1 and G_2 share the parent process graph, there are referred to as single-rate models ($\natural$), otherwise the reaction rate of the graphs can be multi-rate and is indicated by ($\|$). See Example 4.4.8 below for an illustration. For operations with multi-rate executions, min/max delay constraints are considered between all execution events of the respective operations. Verification of such constraints is a difficult problem. Such constraints are not allowed in our formulation of hardware-software cosynthesis.

For operations with single-rate executions, a composite graph model is constructed by merging the respective source and sink vertices of G_1 and G_2 into a single source or sink vertex of G_{12} respectively.

Case B: G_1 and G_2 have a sequential dependency. In this case, a composite constraint graph is constructed either as a serialization from G_1 to G_2 or vice-versa depending upon the ordering relation between G_1 and G_2. A composite constraint graph construction $G_{1;2}$ as a serialization from G_1 to G_2 is carried out by adding an edge from sink of the predecessor graph G_1 to the source of the successor graph G_2. The inter-graph constraints are then added and constraint analysis is carried out on the composite constraint graph model.

Case C: G_1 and G_2 belong to the same hierarchy. Verification of these constraints is carried out by propagating these constraints upwards until these are applicable to the operations in the same graph model.

The following example illustrates the dependencies between flow graphs.

Example 4.4.8. Constraints across flow graphs.

With reference to the graph model hierarchy shown in Figure 30 the following relations are induced:

Case A: Concurrent. $G_{1*}\|G_{2*}$. That is, graphs across two separate process hierarchies may have multiple rates of reaction. No constraints that span across two different hierarchies are supported. Hence constraint analysis disallows use of any constraints that span across flow graphs G_{11} and G_{22}, for instance.

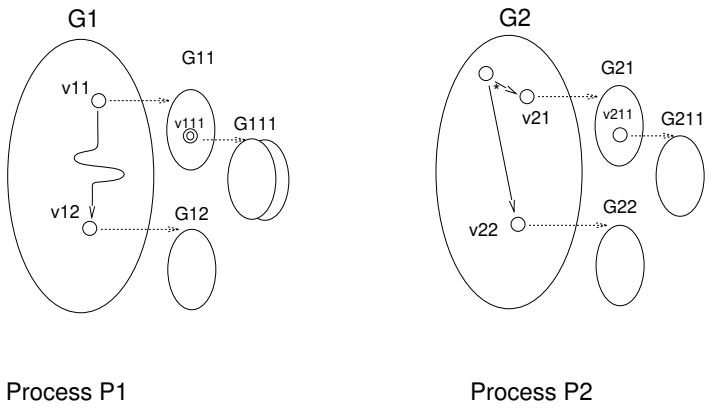


Figure 30: *Graph model hierarchy*

However, it is possible that two graph models belong to the hierarchy of the same process graph and be concurrent. This would be the case when corresponding link operations belong to a *conjoined fork*. For example, consider operations v_{21} and v_{22}. These operations belong to the fanout of a conjoined fork operation and, therefore, these are concurrent. Thus constraints across G_{21} and G_{22} are analysed by the composite constraint graph constructed by composing G_{21} and G_{22} in parallel.

Note that *disjoined forks* lead to mutually exclusive paths, therefore, by definition there can not be constraints on operations that belong to separate conditional paths.

Case B: Sequential. Consider timing constraints that are imposed upon operations in G_{11} and G_{12}. Due to the sequential dependency $v_{11} > v_{12}$, a composite graph is constructed as $G_{11;12}$ and the constraint analysis is carried out on the composite constraint graph model.

Case C: Hierarchical. $G_1 \succ G_{11} \succ G_{111}$ and $G_1 \succ G_{12}$. Similarly for G_2. These graphs belong to the same control hierarchy. Constraints across graph models are considered to be constraints on respective link operations in the parent graph. For instance a constraint that applies to an operation, v_i in G_1 and another operation in G_{11} is treated as a constraint across operations v_i and the link operation v_{11} in G_1.

□

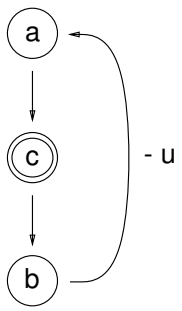


Figure 31: *An $\mathcal{ND}$ cycle in the constraint graph*

4.5 $\mathcal{ND}$ Cycles in Constraint Graph

Figure 31 shows an $\mathcal{ND}$-cycle in the constraint graph. An $\mathcal{ND}$ cycle in the constraint graph is caused either by a maximum delay constraint or by a minimum execution rate constraint. A relative minimum rate constraint on v_i is not propagated above the graph model, G, relative to which it is specified. Therefore, it leads to an $\mathcal{ND}$-cycle only if G^* contains an $\mathcal{ND}$ operation and it is not v_i.

If we consider the constraints as an additional input from the system designer about *properties* of the environment in which the given system operates, then the presence of an $\mathcal{ND}$-cycle in the constraint graph of the system model implies existence of bounds on the delay of the $\mathcal{ND}$ operations. Modeling $\mathcal{ND}$ operations then as purely unbounded operations is restrictive and undermodels the actual design and its environment. We can use constraints to derive bounds on delays of $\mathcal{ND}$ operations. With these bounds, the constraint graph contains operations with bounded delays which is analyzed to determine graph model reaction rates and answer question about satisfiability of constraints at the ports.

4.5.1 Meaning of an $\mathcal{ND}$ cycle

With respect to a $\mathcal{ND}$ cycle Γ there are two possibilities:

- The $\mathcal{ND}$ operation is a *wait* operation. This is also referred to as synchronization-related $\mathcal{ND}$-cycle. The satisfiability of a constraint by a system *implementation* refers to its ability to keep up with a reactive environment under the imposed

constraints. Since the true delay of an $\mathcal{ND}$ operation is determined by the environment, constraint satisfiability for Γ is interpreted as a maximum delay bound on the **active delay** of the operations. In this case, instead of determining the upper bound of delay offered by $\mathcal{ND}$ operation, a **lower bound** on this delay is computed based on implementation choice of Υ. Depending upon how a $\mathcal{ND}$ operation is implemented, the active delay will be related to a *context-switch delay* in software, or in the worst case of *busy-waiting* true delay of the wait operation (as in hardware). In the latter case, constraint satisfiability can only be answered in probabilistic sense.

- The $\mathcal{ND}$ operation is a *loop* operation. (Deterministic) constraint satisfiability in this case requires either an upper bound on the number of times a the loop body can be invoked, **or** a lower bound on how frequently the loop operation is invoked. Fortunately, the latter bound can be determined from the choice of implementation of the graph model containing the loop operation.

The intuitive idea in $\mathcal{ND}$-loop analysis is to use bounds from hardware-software implementations to answer questions about satisfiability of constraints. Due to unconditional invocations of process bodies and busy-wait implementation of $\mathcal{ND}$ operations, purely hardware implementation can not be guaranteed to meet the minimum rate constraints in any deterministic sense. However, for a hardware-software implementation satisfiability of such constraints can be guaranteed under assumptions which are inherent in a mixed implementation, for instance, finite context switch (instead of busy wait) implementation of $\mathcal{ND}$ operations, finite and non-zero delay in runtime scheduling of flow graphs.[2]

Types of loop operation

We examined the semantics of the loop operation, and its implementation based on shared memory and message passing in Example 3.5.10. Here we explore the conditions under which the message-passing implementation of loop operations can be simplified, making it amenable to multirate hardware-software implementation.

[2]Of course, a faster the hardware implementations is always better able to meet the same constraints at higher implementation costs. The essential idea in co-synthesis is to achieve cost-effective implementations while verifiably supporting the performance constraints.

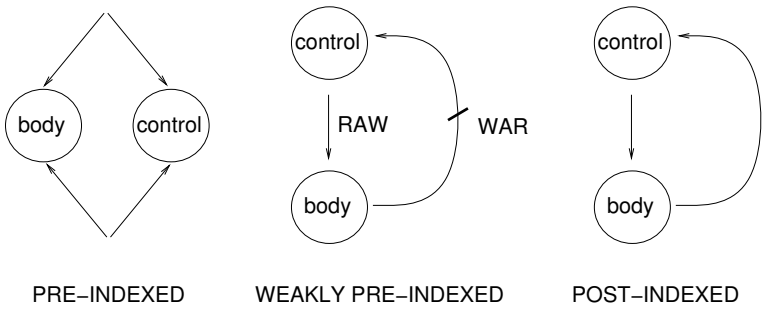


Figure 32: *Types of loop operations.*

A loop body, $G = (V, E)$ consists of two sets of operations: those relating to loop control and those relating to loop body: $V = V_b + V_c$. V_c consists of operations relating to loop condition evaluation, loop register loads and modification of loop index. An execution of loop G consists of a finite number of iterations of V_c and V_b. We assume that each loop is controlled by a single variable index. A loop index value, x, marks execution of loop body until some exit condition becomes true. An operation in G_b either reads the loop index, or writes the loop index or is independent of the loop index. That is, the operations in the loop body can be partitioned into $V_b = V_{br} \cup V_{bw} \cup V_{bn}$ where V_{br} is the set of operations that read the loop index; V_{bw} is the set of operations that modify the loop index and V_{bn} is the set of operations that do not read or modify loop index. With respect to the structure of the loop operations, we now examine three cases (Figure 32).

1. If $V_{br} = V_{bw} = \emptyset$. That is, loop body operations do not affect the loop index. We call these loops **pre-indexed loops**.

2. If $V_{bw} = \emptyset$. That is, loop body operations use but do not modify the loop index. We call such loop **weakly pre-indexed**.

3. If $V_{bw} \neq \emptyset$. That is, the loop index is modified by the loop body. We call such loop as **post-indexed loops**.

Example 4.5.9. Loop types.

Pre-indexed.	*Weakly pre-indexed.*	*Post-indexed.*
read(v);	read(v);	read(v);
repeat{	repeat{	repeat{
write x = y	write x = v	v = read(x);
v = v-1;	v = v-1;	v = v-1;
} until(v);	} until(v);	} until(v);

□

The number of invocations of pre-indexed loops are marked by a loop index variable that is assigned a value at run time but before the execution of the loop body is started. This is in contrast to post-indexed loops where the number of iterations of loop body are determined by the body of the loop operation. For pre-indexed loops depending upon the side effects created by the body of the loop operation, it may be possible to overlap executions of the loop body across invocations of the calling link operation. We consider this possibility in the next section.

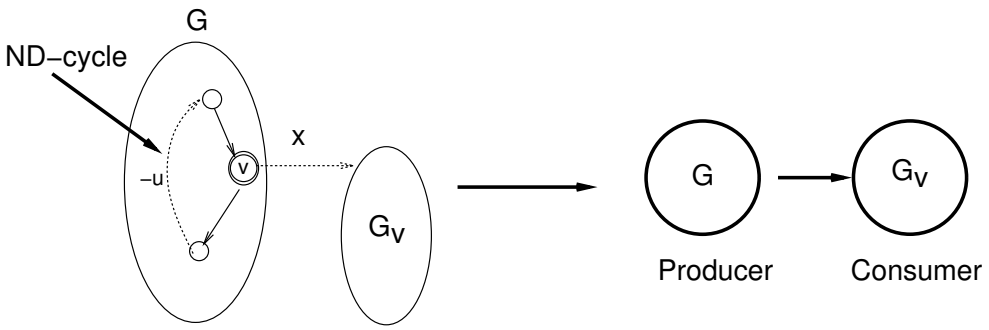


Figure 33: *Modeling an $\mathcal{ND}$ loop as a producer-consumer system*

4.5.2 Problem formulation

Consider the case of a graph model G that contains an $\mathcal{ND}$ loop operation v. The body of the loop operation is modeled by a graph model G_v. Because of the $\mathcal{ND}$ operation v in G, a minimum rate constraint on any operation (other than v) in G will cause an $\mathcal{ND}$-cycle in the corresponding constraint graph of G. In addition, a maximum delay constraint in G may also cause an $\mathcal{ND}$-cycle. We saw in previous sections that constraint satisfiability for $\mathcal{ND}$-cycle leads to a bound, $\overline{x}$, on the number of times the loop body

G_v can be invoked for each invocation of the loop operation, v. We now consider the ways in which this bound can be improved by altering the implementation of the loop operation.

As explained in Example 3.5.9, in general, the loop body G_v consumes some data that is produced by the calling body G and produces some data that is consumed by G. We are interested in cases where the data transfer happens only in one direction, for example, from G to G_v. The data consumed by G_v is defined by the storage that is common to both G and G_v, i.e., $M(G) \cap M(G_v)$. Further, we consider preindexed loops for which the loop index is determined by actual execution of the loop body. Even though given all possibilities such a choice of $\mathcal{ND}$ operation may seem restrictive, from our experience in modeling systems, it defines the most frequent use of loop operations in the hardware descriptions. For such loop operations, as shown in Figure 33, we can think of the called graph model as a consumer and the calling body as a producer. There are various ways of modeling the dynamics of the producer-consumer system. Here we consider a model that explores relationships between the rate of data consumption to the values of input data. Clearly, in general, the rates of data production and consumption are given by the respective reaction rates of the graph models. Let x be the index variable associated with the loop operation, indicating the number of times the loop body G is invoked for an invocation of the loop operation. The fastest rate of production of data by the producer model G is given by the inverse of its minimum latency. This rate of production is fixed by an imposed minimum rate constraint relative to G. The rate of consumption of data, however, is variable and depends upon the actual value of the loop index. That is, the larger the loop index, the longer it takes for the consumer to consume the data.

For a given rate constraint, the bound on the value of the loop index was computed in Section 4.3. In order to maintain correct behavior the producer model must *block* if at any time, the loop index exceeds this bound. This blocking leads to violation of the imposed rate constraint. For this producer-consumer system, since the data transfer occurs only from G to G_v, G need not block for completion of G_v if the loop index is bounded as above. Therefore, we can replace the unknown delay $\mathcal{ND}$ operation, v, by a fixed delay operation which consists in transferring data to a waiting loop body without waiting for completion of the loop operation. Let $\underline{\ell}'(G)$ be the new length vector of G.

For any invocation of loop link operation v, the executions of the loop body G_v must complete before the link operation is restarted. That is,

$$\begin{aligned} & \ell'_m \geq x_v \cdot \ell_M(G_v) \\ \Rightarrow \quad & x_v \leq \frac{\ell'_m}{\ell_M(G_v)} \end{aligned} \tag{4.40}$$

This defines an upper bound on the value of the loop index, x_v.

Definition 4.3 *For a given producer-consumer system the* **blocking limit**, B_1, *is the upper bound on value taken by the loop index beyond which the calling body must block before restarting.*

$$B_1 = \left\lfloor \frac{\ell'_m}{\ell_M(G_v)} \right\rfloor \tag{4.41}$$

B_1 provides a conservative bound on the loop index value based on the fastest rate of production and the slowest rate of consumption of data. We now consider the possibility of extending this bound by altering the structure of the loop operation.

4.5.3 Use of buffers to extend bounds on loop index

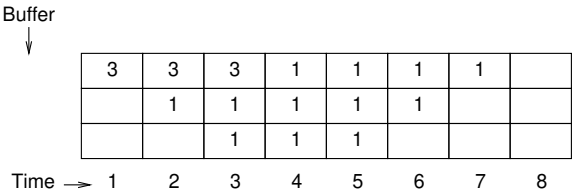
Let us now consider an implementation of the producer-consumer system that is connected by a buffer of depth greater than one. Note that due to the semantics of the loop operation there is always a 1-deep buffer between producer and consumer. In this case, the blocking limit can be extended to

$$B_k = \left\lfloor \frac{k \cdot \ell'_m}{\ell_M(G)} \right\rfloor \tag{4.42}$$

where k is the number of empty spots in the buffer ($\leq$ buffer depth, n). For any loop index value greater than B_1 the execution of consumer model (i.e., body of the loop) spans across successive executions of the link operation in the producer model and, therefore, occupies a place in the buffer. We assume that each invocation of the loop link operation always produces a loop index value greater than or equal to one. That is, it is not the case that an invocation of the loop link operation does not enqueue data into the buffer. This is needed in order to keep the software synchronization simple with low overheads.

Example 4.5.10. Buffering for preindexed loops.

Consider a producer-consumer system, $G - G_v$, where the producer flow graph G produces data at fixed intervals of $\ell'_m = 1$ and maximum path length in the consumer flow graph is $\ell_M(G_v) = 1$. The blocking limit $B_1 = 1$. That is, for each invocation of the producer, the graph G_v can be invoked at most once without having to block G. Assuming 3-deep buffer, the maximum value of loop index can be 3. The following shows a sample execution corresponding to loop index values of $3, 1, 1, 1, 1$:



Note that in hardware-software implementations the buffer between producer and consumer can also be implemented as a serial-parallel *rewording* operation, where the data to be transferred from producer to consumer is reworded as a multiple of original data width. The producer then assembles new words which consists of multiple invocations of the producer. □

Clearly a buffer can help only in conditions where there is irregularity in the values of the loop index and its average value still observes the blocking limit, B_1. In other words, given a finite depth buffer, the producer will always block eventually if the average rate of production is greater than the rate of consumption, that is the average value of loop index exceeds B_1. However, the time it takes to fill up the buffer depends upon the transient behavior of the producer-consumer queueing system. This transient behavior is captured by the following simplification. The producer-consumer system itself is conditionally invoked at a certain rate which is determined by the runtime scheduler or the parent graph model in which the producer-consumer system resides. For each conditional invocation of the producer consumer there is a fixed number of unconditional invocations of the producer-consumer system and at the beginning of each conditional invocation, the producer-consumer system is started from the initial state, that is, all buffers are empty.

Previous work on buffer sizing under rate constraints is by Amon and Borriello [AB91], where the producer-consumer system is modeled as a deterministic queue with

bounds on maximum and minimum rates of data production and consumption. Based on these bounds, an algorithm is presented that first determines a bounded interval over which is queue is guaranteed to be empty (that is, number of productions equals number of consumptions). It then finds a bound on the queue depth based on the transient behavior of the queueing system over this finite interval. The primary difference with the producer-consumer formulation presented here that the queueing system created by $\mathcal{ND}$ cycles is not deterministic, instead the rate of data consumption depends upon the value of the data. It is more appropriately modeled as either as a queueing system with multiple arrivals and fixed rate of consumption or as a system with fixed arrivals with variable rate of consumption. We take the latter approach as described in the following section.

Recently, Kolks *et. al.* [KLM93] have proposed use of implicit state enumeration techniques to determine size of buffers between communicating finite state machines. The procedure is based on representing the buffer as a finite state machine by modeling it as a counter. State reachability analysis on the network of interacting finite state machines is performed to determine the maximum value of the counter used and thus the minimum size of the buffer is determined. This approach is elegant when all parts of a system design can be conveniently modeled as finite state machines. Like the approach in [AB91] it also considers worst case bounds by examining worst case data values.

4.6 Probabilistic Analysis of Min/max and Rate Constraints

So far we have considered only deterministic analysis of constraints and their effect on each other. Such an analysis is carried out by forming algebraic relationships between operation delays, graph lengths and respective min/max delay and rate constraints. Presence of conditional paths and $\mathcal{ND}$ operations in the flow graph model introduces variability in these parameters that limits the scope of a deterministic analysis of the constraints. In this section, we first present a probabilistic analysis of constraints (max delay, min-rate) that lead to creation of $\mathcal{ND}$ cycles in the constraint graph. This analysis

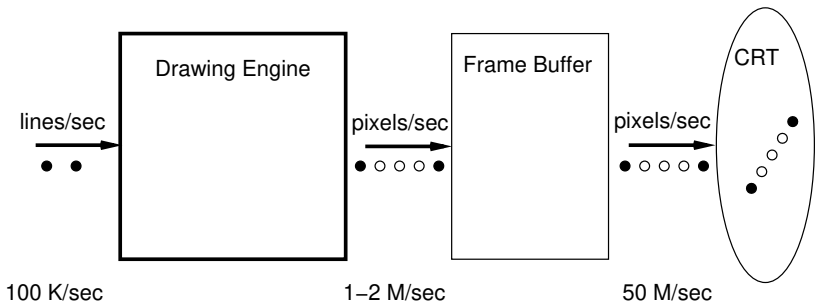
is based on a treatment of the effect of variable delay operations on the operation of the producer-consumer model shown earlier.

Next, a general flow graph consisting of may $\mathcal{ND}$ operations defines a *stochastic process*, that consists of several random variables. We present an analysis of the flow graph to determine operation throughputs for rate constraint analysis by first building an homomorphism between the flow graph representing the stochastic process and a discrete Markov chain. This analysis is carried out to verify marginal satisfiability of the constraints that can not be deterministically satisfiable (Section 4.7).

4.6.1 Meaning of constraint satisfiability

The notion of unbounded delay does not automatically imply infinite delay, but the *possibility* that for any given value, d, the delay offered by the $\mathcal{ND}$ operation can exceed, d. Thus there exists a *distribution* of delay offered by the $\mathcal{ND}$ operation. The situation can be addressed effectively by formulating the notion of possibility of violation with respect to the possibility of exceeding a specified bound on the delay of the $\mathcal{ND}$ operations. Let us consider an example to illustrate the concept.

Example 4.6.11. Constraint satisfiability under $\mathcal{ND}$ cycles. Consider design of



a system for generating pixel coordinates for a line drawing shown in Figure above. The input to the system is a set of coordinates that define the end points of a line. The system generates the pixel coordinates that lie between the two coordinates. The number of pixel coordinates for any given line would depend upon the values of the input coordinates. More pixel coordinates are generated for longer lines.

We are interested in constraints in the input rate (that is, the number of lines per second that the graphics system is able to accept). This rate varies with the input data values. In order to guarantee an absolute bound that is always observed, one

would have to calculate the longest line that can be input and ensure that the system implementation meets the rate constraint under this worst case. An alternative would be to characterize an average case of line lengths, and ensure that the system implementation is able to meet the rate constraint within certain probability of error. □

For a given $\mathcal{ND}$-cycle, a constraint violation occurs if the delay, δ, offered by the $\mathcal{ND}$ operation exceeds a (deterministic) bound, $f(u)$, where δ is a random variable. We define violation error as $\max(\delta - f(u), 0)$. There are various ways to orient the probabilistic analysis. For a given distribution of δ one approach would be to find supportable rate constraints that minimize some measure (absolute, mean-square, etc.) of the violation error.

An alternative approach is to determine supportable rate constraints that contain the probability of constraint violation below some acceptable limit, ϵ. Or, as is possible in the case of preindexed loop operations, find an appropriate size of the buffer that contains probability of a given constraint violation below a given limit. This notion also fits with the general probability of failure for different parts of the system design. In principle, once a constraint violation is brought below a certain error probability that is comparable to probability of failure of other parts of system design the corresponding constraint can be considered **marginally satisfiable**. We take this interpretation to solving satisfiability problem for delay and rate constraints.

For illustration purposes, let us consider a max-delay constraint of Equation 3.17, $t_k(v_j) - t_k(v_i) \leq u_{ij}$. In order for this constraint to be satisfiable this equation must hold true for all values of k. We consider a max-delay timing constraint **marginally satisfiable** if for a given bound $\epsilon, 0 \leq \epsilon \leq 1$, $\Pr\{t_k(v_j) - t_k(v_i) > u_{ij}\} \leq \epsilon$. That is, for each k, the check for constraint satisfiability is considered a trial. A marginally satisfiable constraint is then found to be satisfied if over a large number of such trials, the probability of constraint violation is within a certain specified bound.

Associated with each $\mathcal{ND}$ operation is a variable x that represents the loop index. Recall that for pre-indexed loops, x is computed before invocation of the loop operation. Let us consider x to be a *random variable* that takes value over the set of non-negative integers. The *event space* consists of all possible assignments of the loop index. Let $F_X(\cdot)$ be the probability distribution function associated with random variable x, that is,

F_X is defined over real values such that for a given c, $F_X(c)$ represents the probability that x is below c. In other words,

$$F_X(c) = \Pr\{x \leq c\}$$

The probability density function, $f_X(\cdot)$ is defined as the derivative of the probability distribution function.

Given the loop index of $\mathcal{ND}$ operation as a random variable we formulate the problem of constraint satisfiability as a determination of expected number of data items waiting and thus the size of the buffer required when the loop index has some nonzero probability of exceeding the blocking limit. The answer depends upon the choice of the probability distribution function for the random variable. Based on the distribution of loop index, the buffer depth and probability of buffer being full is developed. A full buffer leads to blocking of the producer process G. The following presents statement of the problem.

> Given a constraint graph model, G with a preindexed $\mathcal{ND}$ cycle Γ caused by a *backward edge* with weight u. Assume that the loop index is a random variable with expected value, $\overline{x}$ and variance σ_x. Let $\underline{\ell}(G_v)$ be the length of the loop body, G_v.
>
> **Problem P1:** *Find a bound N on the buffer size k such that the probability,*
>
> $$\Pr\{G \text{ blocks}\} \leq \epsilon$$
>
> *for all $k \geq N$.*

An alternative formulation of the above problem would be to determine the value of the backward edge that would satisfy blocking limit. This value can then be propagated to determine the achievable execution rate.

> **Problem P2:** *Given a buffer size of k find an upper bound $\overline{u}$ on the weight, u, of the backward edge that causes the preindexed $\mathcal{ND}$ cycle Γ, such that*
>
> $$\Pr\{G \text{ blocks}\} \leq \epsilon$$
>
> *for all $u \leq \overline{u}$. Note that weight of a backward edge is a negative quantity, therefore, an upper bound $\overline{u}$ on u refers to a lower bound on the absolute value of the weight, u.*

Problems P1 and P2 are related. For a given acceptable error rate, probabilistic constraint satisfiability either seeks implementations that meet the required performance (P1) or seeks achievable performance that meets required implementation costs (P2). Note that in the limit $\epsilon \rightarrow 0$ the problems seek a deterministic solution.

4.6.2 Index distribution and bounds on buffer depth

Solution to problems P1 and P2 above depends upon the choice of a probability distribution function, $F_X(\cdot)$ for the random variable x. For analytic simplicity, we treat the random variable as continuous variable and use these to derive approximations to the value of the corresponding discrete parameters. We consider the case when the random variable is unbounded and exponentially distributed. The exponential distribution is chosen due to the fact that the value of the loop index is directly proportional to the interval of time it takes for the consumer to consume a data from the buffer. It has been shown that the exponential distribution has the least information (or highest entropy) and is therefore the most random law that can be used and thus certainly the most conservative approach [CS61]. Additionally, it is the only distribution with the Markovian property. For an exponentially distributed loop index, the rates of data production and consumption follow a Poisson distribution. That is, the times at which data is produced or consumed (i.e., schedule of I/O operations) is *uniformly* distributed. In other words, the k start times of an I/O operation over an interval [0, T] are distributed as the order statistics of k uniform random variables on [0, T].

$$f_e(x) \quad = \quad \mu e^{-\mu x} \tag{4.43}$$

with expected value, $E_e[X] = \frac{1}{\mu}$ and variance $\sigma_e^2 = \frac{1}{\mu^2}$. [3]

[3] Strictly speaking, the distribution should be expressed as $f_e(x)\mu e^{\mu x} \cdot U(x)$ to indicate one-sided nature of the function. In the context of flow graphs, the loop indices are always positive. Therefore, for notational simplicity, we drop the explicit mention of the set function $U()$ and reflect it by adjusting the integration limits.

Lemma 4.3 *For a given error probability, ϵ, the following express the bounds on buffer depth, k*

$$N_e = \left\lceil \frac{\ln \epsilon}{\ln(\frac{B_1}{\overline{x}}) - \ln(e^{B_1/\overline{x}} - 1)} \right\rceil \approx \left\lceil \frac{-\ln \epsilon}{\frac{B_1}{\overline{x}} - \ln(\frac{B_1}{\overline{x}})} \right\rceil \tag{4.44}$$

where $E[X] = \overline{x} < B_1$ is the expected value of the loop index, x and B_1 is the blocking limit for 1-deep buffer.

Proof: Let k be the size of the buffer needed. The bound for the general distribution is obtained by considering k values of the loop index random variable. The k values are assumed to be independent and can be considered as outcomes of k independent trials, or equivalently values of k independent identically distributed (i.i.d.) variables, x_1, x_2, ..., x_k. Now the graph model G blocks if any value of the look index exceeds the blocking limit B_k or the sum over the k variables is greater than B_{2k-1}. We derive an upper bound on the error probability by using the necessary condition to cause blocking if the sum of k i.i.d. variables exceeds the bound $B_k = k \cdot B_1$. Let $y = \sum_{i=1}^{k} x_i$. The distribution of y is given by the convolution of k exponential distributions, each with expected value $\frac{1}{\mu}$. This is shown to be the following Erlangian distribution of type-k [GH74]

$$f_k(y) = \frac{(\mu)^k}{k-1!} x^{k-1} e^{-\mu y} \tag{4.45}$$

For this distribution, the expected value of x is given by $E[Y] = \frac{k}{\mu}$ and variance $\sigma_k^2 = \frac{k}{\mu^2}$.

For a random variable, X, the moment generating function (MGF) is defined as the expected value of e^{tX}, that is, $M_X(t) \doteq E[e^{tX}]$ which is equivalent to Laplace transform (LT) of the probability distribution function, $f_X()$ of X (by substituting parameter t = -s). By the property of Laplace transforms, LT of a convolution of two functions, is a product of their Laplace transforms. Therefore, it can be easily shown that the MGF of a sum of independent random variables is equal to the product of their respective MGFs. In particular,

$$M_Y(t) = E[e^{tY}] = \prod_{i=1}^{k} E[e^{tX_i}] = \left(\frac{\mu}{\mu - t}\right)^k$$

We use this multiplicative property of the MGF in developing a bound on the probability of buffer being full, by using Markov's inequality that $\Pr\{x > a\} \leq E[X]/a$. This bound is then improved by minimizing it using the parameter, t as shown below.

$$\begin{aligned}
\Pr\{G \text{ blocks}\} &= \Pr\{y > k \cdot B_1\} \\
&= \Pr\{e^{ty} > e^{tkB_1}\} \qquad \text{for } t > 0 \\
&\leq \frac{E[e^{ty}]}{e^{tkB_1}} \qquad \text{Markov inequality.} \\
&\leq \min_t \frac{E[e^{ty}]}{e^{tkB_1}} \\
&\leq \left.\frac{E[e^{ty}]}{e^{tkB_1}}\right|_{t = \frac{1}{\overline{x}} - \frac{1}{B_1}} \\
&\leq \left[\frac{\frac{B_1}{\overline{x}}}{e^{\frac{B_1}{\overline{x}}-1}}\right]^k \leq \epsilon
\end{aligned}$$

The result follows by computing the lower bound on k which is defined as N_e. ♯

Note that as $\epsilon \to 0$, $N_e \to \infty$. Figure 34 shows required buffer depth for an $\epsilon = 0.01\%$.

A solution to problem P2 requires determination of an achievable blocking limit, B_1 for a given buffer depth, k. From the blocking limit we can determine the fastest rate of data production, or the maximum value of u and hence the supportable rate constraint. The analytic solution to Equation 4.44, uses Lambert's W function[4] and thus not very useful as a general formula. Instead the Equation 4.44 can be solved numerically using a solver like Maple [Hec93] for the blocking limit, B_1 for a given values of buffer depth $k = N_e$, expected loop index, $\overline{x}$ and the error probability, ϵ.

[4]W function is a solution of the equation $\mathrm{W}(x)e^{\mathrm{W}(x)} = x$.[FSC73]

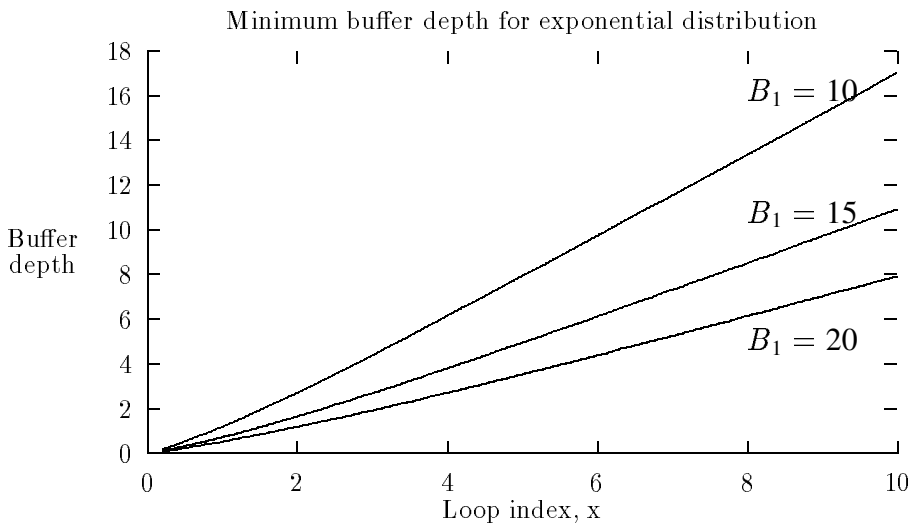


Figure 34: *Buffer depth for exponential distributions* ($\epsilon = 0.01\%$)

4.7 Flow Graph as a Stochastic Process

In the previous section we have seen that a $\mathcal{ND}$-cycle in the constraint graph for a flow graph can be treated as a producer-consumer system, the queueing behavior of which is governed by the values taken by a random variable associated with the loop $\mathcal{ND}$ operation. In this section, we examine a flow graph model that consists of several $\mathcal{ND}$ operations the behavior of each of these operations is governed by a random variable. Thus, a flow graph G with $\mathcal{ND}$ operations corresponds to a **stochastic process**, $\wp$, with the set of random variables, $\{x_i : v_i \in \mathcal{ND}\}$.

$$\wp = \{x_1(t),\ x_2(t),\ \ldots\ , x_n(t), t \geq 0\} \tag{4.46}$$

The process $\wp$ evolves in time as random variables, x, take on actual loop index values. It is easy to see that the random variables share the same event space. Therefore, the random variables x_i are defined over a *common* probability space. We develop the process model by first associating a probability of execution with each vertex in $V(G)$. Recall, from Chapter 3 that at any time a vertex can be in one of the following three states:

reset, *enabled* and *done*. We associate a probability with the transition from an *enabled* state to a *done* state, as defined below

Definition 4.4 Transition probability, p_i *of a vertex,* $v_i \in V(G)$ *is the probability that a currently enabled vertex* i *transitions to a done state in the next cycle (or step).*

This p_i defines the probability that a currently executing node will complete its execution in the next step. Clearly, for a node with a delay, $\delta(v) = 1$ the transition probability is unity. Later in this section we show that the transition probability of a node with delay $\delta \neq 0$ is given by $\frac{1}{\delta}$.

We now consider the behavior of the process $\wp$ over time by transforming into a finite state process where the process is in only one state at any time. We define the state of a process by the set of operations that are enabled at any time. Note that the state of a process is different from the definition of state of a vertex. Due to the concurrency inherent in the flow graph model by means of conjoined forks, there may be more than one operation executing simultaneously. For a flow graph with n vertices, that are a maximum of 2^n possible states. However, the structure of the graph limits this state space based on the partial order on operations. The following example shows possible states in a flow graph.

Example 4.7.12. States in a stochastic flow graph.

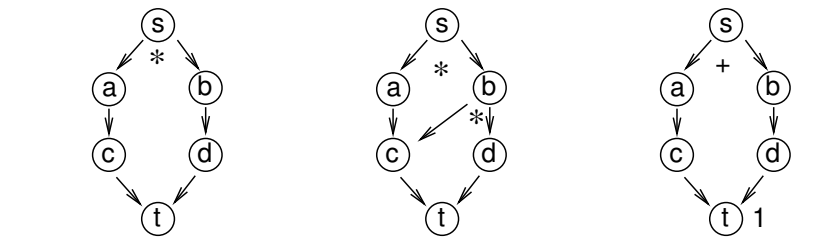


Figure 35: *States of stochastic flow graphs.*

Figure 35 shows flow graphs and possible states. In the first flow graph, due to conjoined fork, the possible states are a product of the states in the two conjoined paths. In the second example, due to additional dependency from 'b' to 'c', the state 'cb' is not possible. Finally, in the third flow graph, due to a disjoined fork, the set of states is the union of states on the mutually exclusive paths. □

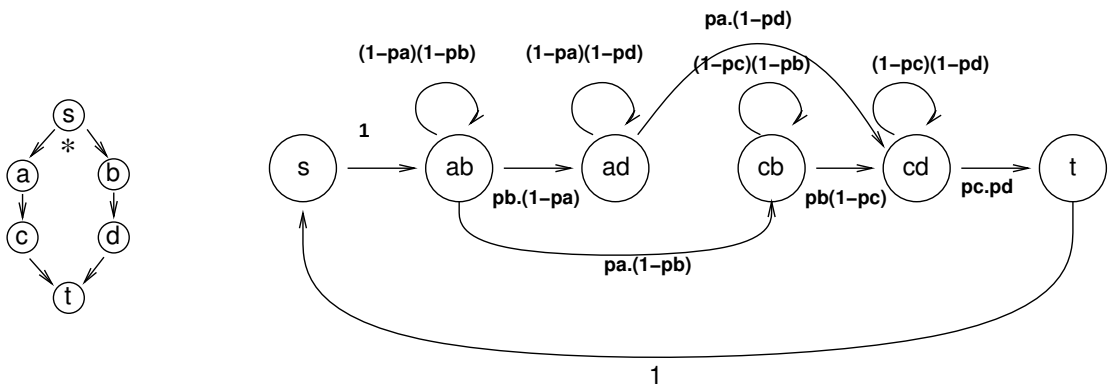
Definition 4.5 *A* **transition process**, $\wp$ *over a graph model,* G *consists of a set of states* $W = \{w_i,\ i = 1, 2, \ldots, |W|\ \}$ *and a set of probabilities,* $\wp = (W,\ \rho)$ *where*

$$\rho_{ij} \doteq \Pr\{W_{k+1} = w_j | W_k = w_i\} \tag{4.47}$$

is the probability from transition from state w_i *to* w_j *in next step.*

The transition probability between two states can be computed from the node transition probabilities. The following shows an example.

Example 4.7.13. Computation of state transition probabilities. The probability



of transition from state 's' to 'ab' (where both 'a' and 'b' are enabled) is 1. The probability of transition from 'ab' to 'ad' is computed by the condition that vertex 'b' changes state from enable to done, p_b, and vertex 'a' stays in the enabled state, $(1 - p_b)$. Since the two events are independent, the probability of state transition is given by the product of individual probabilities. □

We assume that the transitions from a state depend only upon the current state and are independent of the past history of state transitions, that is, the probability of transition from state w_i to w_j is independent of the past history leading up to state w_i.

$$\Pr\{W_{k+1} = w_j | W_k = w_i, W_{k-1} = \ldots\} = \Pr\{W_{k+1} = w_j | W_k = w_i\} \tag{4.48}$$

Assuming the loop index values to be exponentially distributed (which was shown to be the case with some justification in the previous section), we are then able to construct a Markov process from $\wp$.

Definition 4.6 *A Markov chain* **M** *is characterized by a* **transition matrix**, $\mathbf{P} = \{\rho_{ij}\}$ *where*

$$\rho_{ij} = \Pr\{W_{k+1} = w_i | W_k = w_j\}$$

We consider time homogenous Markov changes for which the transition probability is independent of the time step, k.

The Markov chain can be shown to form a positive dynamical system (that is, the state vector also takes on positive values) [Lue79]. It can be shown the largest eigenvalue of **P** is $\lambda_0 = 1$. Constraint analysis using Markov chain attempts to determine the average length of time to reach a specified state. Due to the fact that the flow graph model is a connected graph with a path between every (non-conflicting) vertices (assuming an edge from sink to source operation indicating restart operation), therefore, the Markov chain is regular, irreducible and closed. However, it is not always the case that the chain is aperiodic. But it can be made aperiodic by addition of a variable delay vertex in series from sink to source operations. For mixed implementation using both hardware and software, the software component always contains a runtime scheduler and hence the it can be modeled as a variable delay vertex in series from sink to the source operations. This is, however, not true for purely hardware implementations, where the runtime scheduler operation is zero, and, therefore, the corresponding Markov process may be non-ergodic.

In a regular Markov chain, after a sufficiently large number of steps there exists a nonzero probability of transition between any two states. From the basic limit theorem, for a regular Markov chain the normalized eigenvector of the transition matrix corresponding to its largest eigenvalue determines the steady state occupation probabilities. From ergodic theory, the inverse of the steady state transition probability gives the interval of visitation to the state.

Based on this model, the procedure for constraint analysis of a flow graph model, G, is to first construct a Markov chain **M** by computing all possible states in which the equivalent process model, $\wp$ can be. Recall, a state of $\wp$ at any time is the set of enabled vertices at that time. For this finite state model, we construct the probability of state transitions based on conditional transition probabilities of vertices in a state. The Markov chain **M** is then analyzed to obtain steady state transition probabilities and thus

the expected visitation interval $\mathcal{ND}$ operations. Since a loop $\mathcal{ND}$ operation constitutes a producer-consumer subsystem, these intervals determine the average times between invocations of the producer-consumer subsystems, which are then used to obtain bounds on buffer depths assuming that the producer consumer system is started in the initial state at the beginning of each visit to a state.

Example 4.7.14. Consider the graph model, G shown below. Assume the loop body for each of the three loop operations takes one cycle.

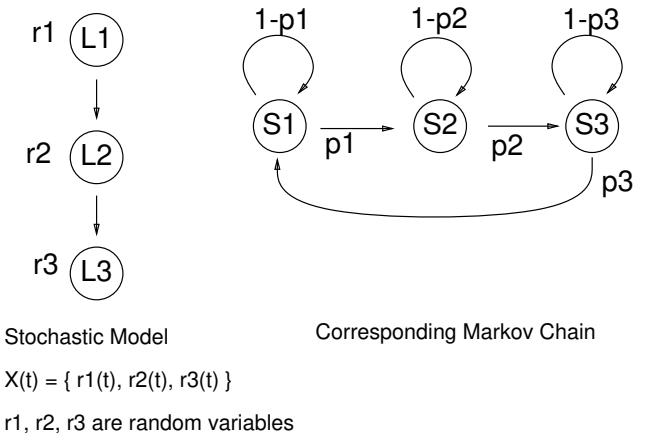


Figure 36: *Loops in a serialized model.*

The corresponding Markov chain is a reachable, closed and irreducible Markov process.

For exponentially distributed random variables with $\overline{r}_1 = 10$, $\overline{r}_2 = 20\,\overline{r}_3 = 50$ the transition matrix of the Markov chain is

$$P = \begin{bmatrix} 0.9 & 0 & 0.02 \\ 0.1 & 0.95 & 0 \\ 0 & 0.05 & 0.98 \end{bmatrix}$$

Steady state probabilities are given by the normalized eigenvector corresponding to the largest eigenvalue, that is,

$$\lim_{m \to \infty} P^m \cdot e_i = [0.125\ 0.245\ 0.625]$$

Where e_i is a column vector with a 1 in the ith row. Now from ergodic theory, the mean recurrence time for a state in the closed Markov chain is inverse of its steady state probability. From this we deduce that steady state interval between end of a

loop to its restart (and therefore, the min-rate of operations in that particular loop) for L_1, L_2, L_3 would be $\frac{10}{0.125} - 10 = 70$, 60 and 30 respectively.

Therefore, for marginally satisfiable rate constraints for G, G_1, G_2 and G_3 would be 80, 70, 60 and 30 cycles/sec respectively. Note that for G, the expected delay is the sum of expected delays of L_1, L_2, L_3. Distribution of the delay is given by the convolution of individual probability distributions. □

Example 4.7.15. Consider loop operations in a fork shown by by the flow graph below. Again assume that loop bodies are one cycle long.

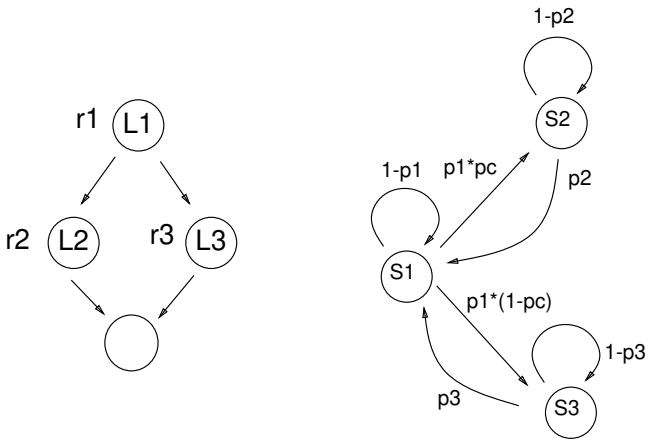


Figure 37: *Loops in a fork.*

In case of a conjoined fork, states S_2 and S_3 belong to the same class. Thus these states are merged before proceeding. In case of a disjoined fork, p_c is the probability of associated conditional transition. For $\overline{r}_1 = 10$, $\overline{r}_2 = 20$ $\overline{r}_3 = 50$, the transition matrix is given by

$$P = \begin{bmatrix} 0.9 & 0.05 & 0.02 \\ 0.1p_c & 0.95 & 0 \\ 0.1(1-p_c) & 0 & 0.98 \end{bmatrix}$$

The steady state probabilities are [0.22 0.22 0.56] and [0.208 0.167 0.625] for $p_c = 0.5$ and $p_c = 0.4$ respectively.

Therefore, marginally satisfiable minimum rates are as follows:

Graph model	$p_c = 0.4$ (cps)	$p_c = 0.5$ (cps)
G_1	30.08	35.45
G_2	99.76	70.91
G_3	30	49.29
G	55.71	55.71

Note that latency of G is expressed as $r_1 + \max(r_2, r_3)$. Its distribution function is $f_{R1} \star (f_{R2}F_{R3} + f_{R3}F_{R2})$ with expected value $= 20 + 50 - \frac{1}{\frac{1}{20}+\frac{1}{50}} = 55.71$. Here $\star$ is the convolution function. □

Limitations and possible extensions

The stochastic analysis of graph models is not new and has been carried out before by several researchers for different types of network models. For related work on timed and stochastic Petri nets see for example [Sha79] [Mol82]. The chief limitation is in the construction of the possible states for the process which requires enumeration of all possible paths of execution, and this can lead to explosion in the number of states in **M**. Further, the assumption of exponential distribution of loop index values may be fine for truly 'random' loop indices. However, for loops with indices that have deterministic relationships, this assumption is harder to justify. Aperiodicity of the constructed is not always guaranteed. However, it can be guaranteed by adding a dummy $\mathcal{ND}$ operation whose corresponding state has a path to every other state in the Markov chain, and therefore, it belongs to the same class as others in **M** and is aperiodic, therefore, making all states in **M** aperiodic.

A large number of states in a Markov chain makes it harder to analyze the resulting stochastic matrix for the steady state behavior. The following transformation to collapse a chain of states in the Makov process can be used to achieve reduction in the number of states. This reduction is based on a notion of equivalence of Markov chains such that for a given state in the two chains, the steady state occupation probabilities are the same. The following defines the concept.

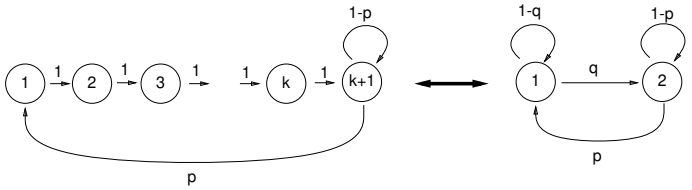
Definition 4.7 *Markov processes* $\mathbf{M_1}$ *and* $\mathbf{M_2}$ *are considered* homomorphic in steady-state, (HSS)*, if steady state transition probabilities of states common in* $\mathbf{M_1}$ *and* $\mathbf{M_2}$ *are identical.*

Theorem 4.6 [State reduction] *A chain of states,* $s_1,\ s_2,\ \ldots, s_k$ *with transition probabilities*

$$\rho_{ij} = \begin{cases} 1 & j = i+1 \\ 0 & \text{otherwise} \end{cases}$$

can be reduced to a single state with probability of exiting the state $= \frac{1}{k}$.

Proof: Consider $\mathbf{M_1}$ and $\mathbf{M_2}$ with transition matrices:



$$P_1 = \begin{bmatrix} \mathbf{0}_k^T & \vdots & p \\ \cdots & \cdots & \cdots \\ \mathbf{I}_k & \vdots & \mathbf{0}_{k-1} \\ & \vdots & 1-p \end{bmatrix} \tag{4.49}$$

$$P_2 = \begin{bmatrix} 1-q & p \\ q & 1-p \end{bmatrix} \tag{4.50}$$

where $\mathbf{0}_k$ is zero column vector of dimension k and $\mathbf{I}_k$ is a $k \times k$ identity matrix.

$$\mathbf{M_1} \text{ and } \mathbf{M_2} \text{ are HSS } \Leftrightarrow e_1^T \cdot P_1^\infty \cdot e_{k+1} = e_1^T \cdot P_2^\infty \cdot e_2 \tag{4.51}$$

where e_i is a column vector with a 1 in the ith place, rest being zero. The largest eigenvalue of P_1 and P_2 are 1. The eigenvectors corresponding to this eigenvalue for P_1 and P_2 are

$$\begin{aligned} v_1 &= [1\ 1\ \cdots\ 1\ \frac{1}{p}] \\ v_2 &= [1\ \frac{q}{p}] \end{aligned}$$

The steady state probabilities of the state $k+1$ in $\mathbf{M_1}$ and 2 in $\mathbf{M_2}$ are given by the normalized eigenvector. That is,

$$\begin{aligned} e_1^T \cdot P_1^\infty \cdot e_{k+1} &= \frac{1/p}{k + \frac{1}{p}} \\ e_1^T \cdot P_2^\infty \cdot e_2 &= \frac{q/p}{1 + \frac{q}{p}} \end{aligned}$$

Therefore,

$$\mathbf{M_1} \text{ and } \mathbf{M_2} \text{ are HSS} \quad \Leftrightarrow \quad q = 1/k$$

♯

4.8 Summary

We have developed several notations and concepts in this section. Let us summarize the main points. The notion of constraint satisfiability is developed based on the ability to discern *existence* a potential schedule of operations that meets the constraint.

For reasons of simplicity, scheduling is considered in two parts: operation scheduling (or the short term scheduling) and task scheduling (or the long term process scheduling). While the former can be subject to deterministic constraint satisfiability analysis, such analysis for the latter is limited in applicability due to the additional non-determinism inherent in this kind of scheduling. We attempt to capture this non-determinism and analyze it together with the non-deterministic delay operations.

The run-time scheduler models uncertainty in invocation of graph models and thus attempts to merge this uncertainty with that of delay of $\mathcal{ND}$ operations. This 'merge' in uncertainty is accomplished by redefining short-term constraint satisfiability over *active* computation times rather than total execution times. Thus a $\mathcal{ND}$ operation is transformed into a fixed-active-delay operation while the uncertainty associated with its actual delay is delegated to the runtime (or long term) scheduler. Since the idea of active computation time is naturally suited to a software execution environment the implementation of two step scheduler is restricted only to software. However, it is conceivable that with appropriate control generation scheme this idea can be used for hardware implementation as well.

The data-dependent delay operations are similarly transformed into fixed delay operations by means of buffers. Probabilistic analysis is used to determine likelihood of constraint violation based on probability of buffer overflow. It is reasoned that such analysis is more relevant to answering satisfiability questions of constraints that involve application of the long term scheduler. The probabilistic analysis is done by constructing a probabilistic producer-consumer system whose behavior is controlled by a random

variable. Next, we construct a stochastic process by collecting a number of random variables each corresponding to non-deterministic delay operation. This process is then transformed into a stationary Markov chain. The Markov chain is analyzed to determine steady state probabilities. Then using results from ergodic theory we find average interval between visits to various states which is used to determine satisfiability of constraints on min-rate of the corresponding flow graph model.

There are several limitations of the producer-consumer formulation on which the probabilistic analysis is based upon. Firstly, use of buffer in case of preindexed loops *alters* the input/output behavior of the modeled system. Specifically, the events at system ports are reordered such that the sequence of events at any port still follow the same order, however, events across ports may be interleaved. In other words, if we consider a port operation to be an action and specific instances of execution to be events, the partial order imposed by the system model is the order on actions and not on events. Intuitively this altered behavior would be acceptable as long as the environment contains no interactions between separate input/output events (for example synchronization). This may not be true in general for all systems. As safe approach would be to construct buffered producer-consumer systems only for those systems where input/output operations do not contain any explicit sequencing dependencies (i.e., only inter-iteration dependencies).

In this dissertation we concern ourselves only to systems where operations and dependencies are static. In general, long term scheduling also depends on dynamic and runtime factors and information needed for an efficient schedule is available only at runtime. These runtime factors include data-dependent dependencies between operations, data availability and synchronization. These complexities make dynamic scheduling problem difficult to formulate and analyze. In practice heuristic solutions are sought for solving such problems. The biggest drawback of runtime scheduling methods is performance loss due to the overheads. Dynamic scheduling techniques are out of the scope of this research work.

Chapter 5

Software and Runtime Environment

In this chapter we focus on the problem of synthesis of the software component of system design. We consider the software portion to be of limited size and mapped to real memory so that the issues related to virtual memory management are not relevant to this problem. The objective of software implementation is to generate a sequence of processor or machine instructions from the set of flow graph models. Due to significant differences in processor abstractions at the levels of graph model and machine instructions, this task is performed in two separate steps: (1) generation of a *program* in a high-level programming language, C; followed by (2) compilation of the program into machine instructions by software compiler and assembler for the processor. We assume that the processor is a predesigned general-purpose component with available compiler and assembler. Therefore, the important issue in software synthesis is generation of the source-level program. Most of this chapter is devoted to this step of software synthesis. Towards the end of this chapter, we discuss important issues related to software compilation and linking.

This chapter is organized as follows. We first present a cost model of the processor in Section 5.1. In Section 5.2 we present a model of software that allows for satisfaction of timing constraints for a target architecture that consists of a single processor. Estimation of software performance in view of the cost model of the target processor is presented in Section 5.3. Software performance is also affected by the allocation and management of storage for program and data portions of the software. Estimation of software storage and its effect on performance is discussed in Section 5.4. Section 5.5 presents an overview

of the steps in synthesis of the software component. These steps are presented in detail in Sections 5.6, 5.7 and 5.8. Section 5.9 presents practical issues in generation of the software. We conclude this chapter by a summary in Section 5.10.

5.1 Processor Cost Model

An implementation of a flow graph in software is characterized by assignment of delays to operation vertices and choice of a runtime scheduler. The delay of an operation is dependent on the set of processor instructions and delays associated with these instructions. The instruction set and associated delays are determined by the choice the processor. We capture processor specific information into a cost model described in this section. The intermediate format for capturing this cost model is presented in Appendix C.

Processors are the *predesigned* hardware components that execute general-purpose software. To a compiler, a processor is characterized by its instruction set architecture (ISA) which consists of its instructions and the memory model. We make following assumptions on the ISA:

- The processor is a general purpose *register* machine with only explicit operands in an instruction (no accumulator or stack). All operands must be named. This refers to the most general model for code generation. General purpose register machines is the most dominant architecture in use today and is expected to remain so in forseeable future [HP90].

- The memory addressing is based on *byte-level addressing*. This is consistent with the prevailing practice in the organization of general-purpose computer systems.

A processor instruction consists of an operation and a set of operands on which to perform the specified operation. While the actual instruction sets for different processors are different, a commonality can be established based on the *types* of instructions supported. For our purposes we assume a *basic* set of instructions listed in Table 3. This set of basic instructions groups together functionally similar operations. In addition, it also contains *macro*-operations that may not be available as single instructions, for example,

call and return. These operations help in software delay estimation by providing additional information which may not be available purely from looking at the instruction set of a processor.

There is a significant variation in the types of operands supported on different processors. Following the taxonomy in [HP90] we classify ISA in to following categories:

Load-Store (LS): This refers to an ISA where memory operations are confined to two types of instructions (LOAD) and (STORE). All other instructions use non-memory operands.

Register-Memory (RM): In this ISA, all instructions may have up to one memory operand.

Memory-Memory (MM): All operands in an instruction may be memory operands.

Based on this understanding of processor and instruction set architecture we develop a **cost model** to represent the target processor,

$$\Pi = (\tau_{op},\ \tau_{ea},\ t_m,\ t_i) \tag{5.52}$$

where the

- Execution time function, τ_{op}, represents instruction delay times in cycles for operations listed in Table 3,
- Address calculation delay function, τ_{ea}, represents effective address calculation delay times in cycles,
- Memory access time t_m is the time in cycles for a memory access. Note that t_m can also be used to model a memory access with wait states,
- Interrupt response time, t_i, is the maximum time between the activation of an external interrupt and the beginning of execution of the corresponding interrupt service routine.

Instruction type	*Meaning*	*DLX Example*
load	Load from memory	lb, lbu, lh, lhu, lw, lwl, lwr, *la, *li
store	Store to memory	sb, sh, sw, swl, swr, *ulh, *ulhu, *ulw, *ush, *usw
move	Move registers	mfhi, mthi, mflo, mtlo, *mov
xchange	Exchange registers	–
alu	ALU operations	addi, addiu, andi, ori, xori add, addu, sub, subu, and, or, xor, nor sll, srl, sra, sllv, srlv, srav lui, *abs, *neg, *negu, *not, *rol, *ror *seq, *sle, *sleu, *sgt, *sgtu, *sge, *sgeu, *sne
mpy	Integer multiply	mult, multu
div	Integer divide	div, divu, *rem, *remu
comp	Compare	slti, sltiu, slt, sltu
call	Call	–
jump	Jump	j, jal, jr, jalr
branch	Branch	beq, bne, *bgt, *gge, *bgeu, *gbtu, *blt, *ble, *bleu, *bltu
bc_true	Branch taken	–
bc_false	Branch not taken	–
return	Call return	–
seti	set interrupt	–
cli	clear interrupt	–
int_response	Interrupt response	–
halt	Halt	–
		* = *synthesized instruction.*

Table 3: *Basic instruction set*

The execution time function, τ_{op}, maps *assembly language instructions* to positive integer delays. The assembly language instructions are generated by the high-level language compiler. These instructions usually correspond to instructions supported by the processor instruction set. However, some assembly language instruction may refer to a group of processor instructions. These *macro-assembly* language instructions are sometimes needed for compilation efficiency and to preserve the *atomicity* of certain operation in the flow graph model. The effect of internal hardware pipelining in microprocessors is modeled as follows. The function, τ_{op} represents a pipelined operation delay (which is usually 1 cycle for most operations). A penalty of $p - 1$ cycles is added to the delay of the overall program. In addition, a *pipeline stall* penalty is added for instructions with latencies greater than p. The interrupt response time, t_i, is the time that processor takes to become aware of an external hardware interrupt in a single interrupt system (that is, when there is no other maskable interrupt is running). For cost model description purposes, this parameter can be specified as a part of the operation delay function (as shown by entry `int_response` in Table 3).

The address calculation function, τ_{ea}, maps a memory addressing mode to the integral delay (in cycles) encountered by the processor in computing the effective address. An addressing mode specifies an immediate data, register or memory address location. In the last case, the actual address used to access the memory is called the effective address. Table 4 lists common addressing modes. Square brackets ([]) indicate contents, for example, [R1] indicates contents of register R1, mem[10] indicates the contents of memory at address 10.

For completeness sake, this table lists addressing modes that are encountered in general programs. However, when generating programs from HDL descriptions some of these modes are never used. For example, a computed reference (register indirect) usually occurs when the data value is created dynamically or a local variable (stack) is referred to by means of a pointer or an array index. Neither of these conditions occur when generating code from HDL. Further, not all the addressing modes may be supported by a given processor. For example, the DLX processor supports only immediate and register addressing modes, while the x86 instruction set supports all mentioned addressing modes

Mode	*Notation*	*Explanation*	*Usage*
immediate	#4	value = 4	Constants
register	R1	value = [R1]	Register values
direct	(100)	value = mem[100]	Static data
register indirect	(R1)	value = mem[[R1]]	Pointer/computed address
register offset	(40)R1	value = mem[[R1]+40]	Local variables
memory indirect	@(R1)	value = mem[mem[[R1]]]	Pointer access
indexed	100(R1)(R2)	value = mem[100+[R1]+d*[R2]]	Array elements

Table 4: *Addressing modes*

(though with restrictions on which registers can be used in a certain addressing mode).

Storage alignment

Storage alignment is a side-effect of the byte-level addressing scheme assumed for the processor/memory architecture. Because the smallest object of a memory reference is a byte, references to objects smaller than a byte must be aligned to a byte. Further, for memory efficiency reasons, the objects that occupy more than a byte of storage are assigned an integral number of bytes, which means their addresses must also be *aligned*. For example, address of a 4-byte object (say integer) must be divisible by 4.

Table 5 lists data types and alignment requirements which are taken into account in the determination of the data size. The size of a *structure* is determined by the total of size requirements of its members. In addition, the structure must *end* at an address determined by the most restrictive alignment requirement of its members. This may result in extra storage (upto a maximum 3-bytes per member) for padding. In the case of a structure consisting entirely of bit fields, there is no padding if the total number of bits is less than 32 bits. In case of structure widths greater than 32 bits, additional 32-bit words are assigned and members that lie on the boundary of two words are moved to the subsequent word leaving a padding in the previous word. It is assumed that no member takes more than 32-bits. Variables with size greater than 32-bits, are bound to multiple variables represented by an array. The size and alignment requirements are then multiplied by the number of array elements.

Data type	*Size*	*Address alignment*	*Unsigned range*	*Signed range*
int	32	%4	$-2^{31} \ldots 2^{31}-1$	$0 \ldots 2^{32}-1$
short	16	%2	$-32,768 \ldots 32,767$	$0 \ldots 65,535$
char	8	%1	$-128 \ldots 127$	$0 \ldots 255$
pointer	32	%4	NA	$0 \ldots 2^{32}-1$
struct	*variable. see text.*			

Table 5: *Variable types and storage*

Example 5.1.1. Variable storage assignments.

The following shows the set of variables used in the definition of a flow graph and the corresponding storage assignments in the software implementation of the graph.

```
a[1], b[2], c[3], d[4], e[5]        struct{ a:1; b:2; c:3; d:4; f:5 }
f[33]                               int f[2]
```

Minimum storage used in the flow graph model is 8 bytes. However, due to alignment requirements the actual data storage is 12 bytes. □

5.2 A Model for Software and Runtime System

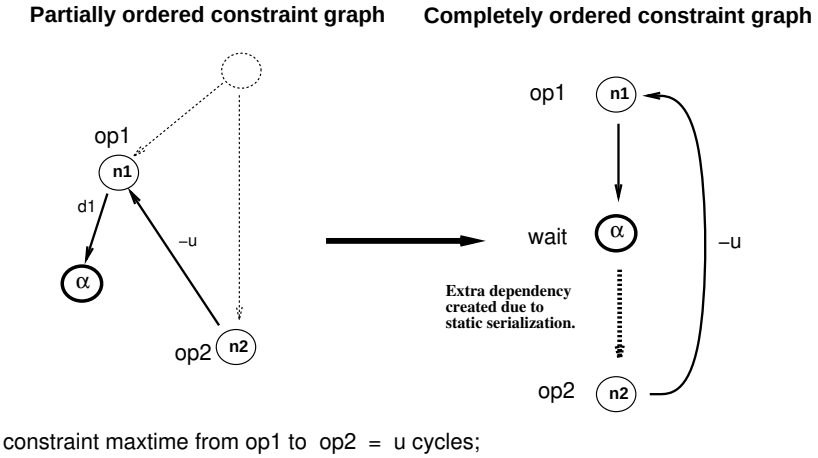
The concept of a runtime system applies to systems containing a set of operations or tasks and a set of resources that are used by the tasks. Operations may have dependencies that imposes a (partial) ordering in which the tasks can be assigned to resources. In general a runtime system consists of a *scheduler* and a *resource manager*. The task of the runtime scheduler is to pick up a subset of tasks from the available set of tasks to run at a particular time step. The resource manager can be thought of consisting of two components: a resource *allocator* and a resource *binder*. The allocator assigns a subset of resource to a subset of tasks, whereas a binder makes specific assignments of resources to tasks. The results of the scheduling and resource management tasks are interdependent, that is, a choice of a schedule affects allocation/binding and vice versa. Depending upon the nature and availability tasks and resources some or all of these activities can be done

either *statically* or *dynamically*. A static schedule, allocation or binding make the runtime system simpler.

In this general framework, most synthesized hardware uses static resource allocation and binding schemes, and static or relative scheduling techniques as described in Chapter 4. Due to this static nature, operations that share resources are serialized and the binding of resources is built into the structure of the synthesized hardware, and thus there are always enough resources to run the available set of tasks. Consequently, there is no need for a runtime system in hardware. Similarly, in software, the need for a runtime system depends upon the whether the resources and tasks (and their dependencies) are determined at compile time or runtime.

Since our target architecture contains only a single resource, that is, the processor, the tasks of allocation and binding are trivial, i.e., the processor is allocated and bound to all routines. However, a *static* binding would require determination of a *static* order of routines, effectively leading to construction of a single routine for the software. This would be a perfectly natural way to build the software given the fact that both resources and tasks and their dependencies are all statically known. However, due to the presence of $\mathcal{NTD}$ operations in software, a complete serialization of operations may lead to creation of $\mathcal{NTD}$ cycles which would make satisfiability determination impossible.

Example 5.2.2. Static linearization leads to creation of $\mathcal{NTD}$ cycles.



Consider a part of the flow graph shown in figure above. Any ordering of operations op1 and op2 that puts an $\mathcal{NTD}$ operation α between these two operations creates an

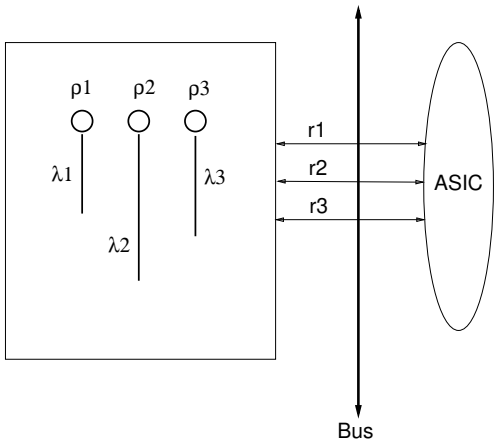


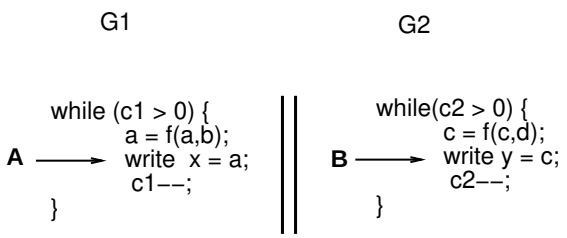
Figure 38: *Software model to avoid creation of $\mathcal{ND}$ cycles.*

$\mathcal{ND}$-cycle in the corresponding constraint graph. □

A solution to this problem is to think of software as a set of **concurrent program threads** as sketched in Figure 38. A thread is defined as a linearized set of operations that may or may not begin by an $\mathcal{ND}$ operation. Other than the beginning $\mathcal{ND}$ operation, a thread does not contain any $\mathcal{ND}$ operations. The latency of a thread is defined as sum of the delay of its operations without including the initial $\mathcal{ND}$ operation whose delay is merged into the delay due to the runtime scheduling function, Υ.

The use of multiple concurrent program threads instead of a single program to implement the software avoids the need for complete serialization of all operations which may create unbounded cycles. In addition it also makes it possible to share resources, in this case the processor, dynamically.

Example 5.2.3. Consider the concurrent execution of following two graphs in a system model.: Here the function f() represents call a resource that is common



to G_1 and G_2. "A" and "B" are operation tags on the write operations in the two

graphs. Assume that operations, "A" and "B" are rate constrained. Let us consider the trace of execution for operations "A" and "B". A static resource allocation strategy between G_1 and G_2 requires serialization between G_1 and G_2 since there is only one resource available to implement the function f(). This serialization leads to either of the following traces:

$$\text{A, A, A, } \ldots \text{, A, B, B, B, } \ldots \text{, B}$$

or

$$\text{B, B, B, } \ldots \text{, B, A, A, A, } \ldots \text{, A}$$

On the other hand, when using dynamic allocation of the processor to two program threads corresponding to G_1 and G_2, it possible to obtain following trace of execution:

$$\text{A, B, A, B, A, B, } \ldots,$$

where the interval between consecutive A and C operations can be determined statically and, therefore, it can be constrained. This allows us the possibility to support execution rate constraints on the write operation at ports x and y. In the former case, due to static serialization of G_1 and G_2 an execution rate constraint would lead to an $\mathcal{ND}$-cycle that would only be marginally satisfiable by bounding the loop indices of the two loop operations. By using dynamic resource allocation, the rate constraints can now be satisfied deterministically. □

In this model of software, satisfiability of constraints on operations belonging to different threads can checked for marginal satisfiability as defined in Chapter 4 (that is, assumed a bounded delay on scheduling operations associated with $\mathcal{ND}$ operations). Constraint analysis for software depends upon first arriving at an estimate of the software performance and size of register/memory data used for the software. We discuss these two issues next.

5.3 Estimation of Software Performance

A program is compiled into a sequence of machine instructions. Therefore, timing properties of a program are related to the timing properties of the machines instructions to which it is eventually translated. Any variability in machine instruction timings is reflected on the variability of timing of programming-language statements. One approach to software estimation would be to generate such estimates directly from synthesized and compiled

machine instructions for a given graph model. However, the process of compilation of high-level language programs is time intensive and may not be a suitable step when evaluating tradeoffs among software and hardware implementations.[1] Therefore, alternative methods are sought for estimating software timing properties directly from programs written in high-level languages. When deriving timing properties from programs, several problems are encountered due to the fact that popular programming languages provide an inherently asynchronous description of functionality, where the program output is dependent on the timing behavior of its components and of its environment. Attempts have been made to annotate programs with relevant timing properties [Sha89, Mea89]. Syntax-directed delay estimation techniques have been tried [PS90] which provide quick estimates based on the language constructs used. However, syntax-directed delay estimation techniques lack timing information that is relevant in the context of the semantics of operations.

We perform delay estimation on flow graph models for both hardware and software, using the semantics interpretations of operations in our estimation procedures. A software delay consists of two components: delay due operations in the flow graph model, and delay due to the run-time environment. We discuss the run-time environment on constraint satisfiability in the next chapter. Here we focus on the first component, that is, the delay of a software implementation of the operations in the flow graph model. For this purpose, it is assumed that a given flow graph is to be implemented as a single program thread. Multiple program thread generation is achieved similarly by first identifying subgraphs corresponding to program threads. This identification of subgraphs is discussed in next chapter. Software delay then depends upon the delay of operations in the flow graph model *and* operations related to storage management.

Calculations of storage management operations is described in Section 5.4.2.

5.3.1 Operation delay in a software implementation

In order to make effective tradeoffs during partitioning, it is necessary to be able to make good estimates about software and hardware performance. Such estimates often require

[1] Vulcan-II does provide commands to perform exact software delay estimations based on direct synthesis and compilation. See Chapter 7.

simplifying assumptions that tradeoff modeling accuracy against speed. In estimating software performance, we make the following assumptions.

1. The system bus is always available for instruction/data reads and writes.

2. All memory accesses are aligned.

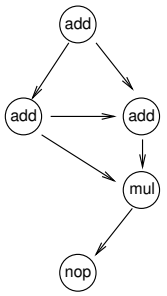
3. All memory accesses are to a single-level memory.

Each operation v in the flow graph is characterized by a number of read accesses, $m_r(v)$, a number of write accesses, $m_w(v)$ and a number of assembly-level operations, $n_o(v)$. The software operation delay function, η, is computed as follows:

$$\eta(v) = \sum_{i=1}^{n_o(v)} t_{op_i} + (m_r(v) + m_w(v)) \times m_i \tag{5.53}$$

where the operand access time, m_i, is the sum of effective address computation time and memory access time for memory operands. For some instruction-sets, not all possible combinations of ALU operations and memory accesses are allowed and often operand access operations are optimized and overlapped with ALU operation executions thus reducing the total execution delay. Due to this non-orthogonality in ALU operation execution and operand access operations, the execution time function of some operations is often overestimated from real execution delays. In the one-level memory model, the number of read and write accesses depends upon the fanin and fanout of the operation.

Example 5.3.4. Software delay estimation.



For the graph model shown above, assuming addition delay 1 cycle, multiplication delay is 5 cycles and memory delay 3 cycles.

Assuming that each non-NOP operation produces a data, that is, $m_w(v) = 1$ and that the number of memory read operations are given by the number of input edges, the software delay associated with the graph model is $3 \times t_+ + t_* + (5+4) \times m_i = 35$ cycles. □

Use of operation fanin and fanout to determine memory operations provides an approximation for processors with very limited number of available general purpose registers. Most processors with load-store (LS) instruction set architectures feature a large number of on-chip registers. Therefore, we develop a model of register usage in these processors in Section 5.4.2. Based on this model, we determine the memory access operations and their contribution to the software delay.

$\mathcal{ND}$ operations

Wait operations in a graph model induce a synchronization operation in the corresponding software model. Thus, the software delay of wait operations is estimated by the *synchronization overhead* which is related to the program implementation scheme being used. One implementation of a synchronization operation is to cause a *context switch* in which the waiting program thread is switched out in favor of another program thread. It is assumed that the software component is computation intensive and thus the wait time of a program thread can be overlapped by the execution another program thread. After the communication operation associated with the wait operation is complete, the waiting program thread is resumed by the runtime scheduler using any specific scheduling policy in choosing among available program threads. Alternatively, the completion of communication operation associated with wait operation can also be indicated by means of an interrupt operation to the processor. In this case, the synchronization delay is computed as follows:

$$\eta_{intr}(v) = t_i + t_s + t_o \tag{5.54}$$

where t_i is interrupt response time, t_s is interrupt service time, which is typically the delay of the service routine that performs input read operation and t_o is concurrency overhead. The notion of concurrency overhead is discussed in Section 5.7.

Link operation are implemented as call operations to separate program threads corresponding to bodies of the called flow graphs. Thus the delay due to these operations is

accounted for as the delay in implementation of control dependencies between program threads as discussed in Section 5.6.

5.4 Estimation of Software Size

A software implementation of a flow graph model, $G = (V, E)$ is characterized by a software size function, S^{Π}, that refers to the size of program, S_p^{Π}, and static data, S_d^{Π}, necessary to implement the corresponding program on a given processor, Π. For an system model, Φ,

$$S^{\Pi}(\Phi) = \sum_{G_i \in \Phi} S^{\Pi}(G_i) = \sum_{G_i \in \Phi} [S_p^{\Pi}(G_i) + S_d^{\Pi}(G_i)] \tag{5.55}$$

The set S_d^{Π} consists of storage required to hold variable values across operations in the flow graph and across the machine operations. This storage can be in the form of specific memory locations or the on-chip registers, since no aliasing of data items is allowed in input HDL descriptions[2]. In general, $S_d^{\Pi}(G)$ would correspond to a subset of the variables used to express a software implementation of G, that is,

$$S_d^{\Pi}(G) \leq |M(G)| + |P(G)| \tag{5.56}$$

where $M(G)$ refers to the set of variables used by the graph G and $P(G)$ is the set of input and output ports of G (as defined in Chapter 3). This inequality is because not all variables need be *live* at the execution time of all machine instructions. At the execution of a machine instruction, a variable is considered live if it is input to an future machine instruction. Interpretation of variables in relation to flow graph is discussed in Section 5.4.2.

In case $S_d^{\Pi}(G)$ is a proper subset of the variables used in software implementation of G, that is, $M(G)$, additional operations (other than the operation vertices in G) are needed to perform data transfer between variables and their mappings into the set $S_d^{\Pi}(G)$. In case $S_d^{\Pi}(G)$ is mapped onto hardware registers, this set of operations is commonly referred to as *register assignment/reallocation operations*. Due to a single-processor target

[2]Register storage of an aliased variable will lead to incorrect behavior due to possible inconsistency in values stored in the register and the value stored at the memory.

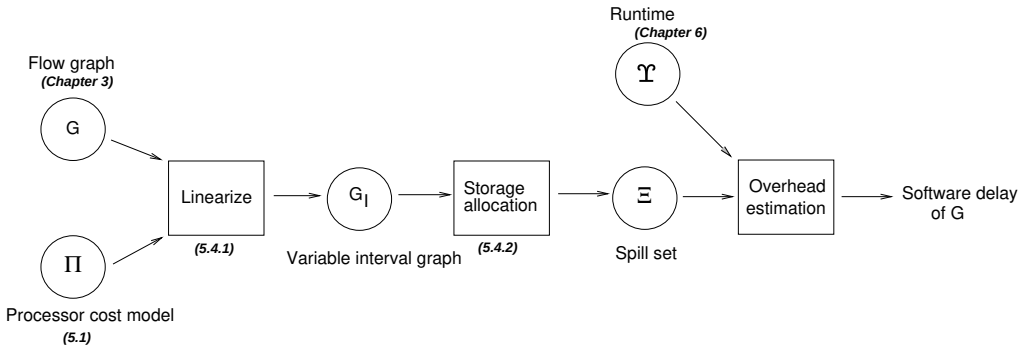


Figure 39: *Software delay estimation flow.*

architecture, the cumulative operation delay of $V(G)$ would be constant under any schedule. However, the data set $S_d^{\Pi}(G)$ of G would vary according to scheduling technique used. Accordingly, the number of operations needed to perform the requisite data transfer would also depend upon the scheduling scheme chosen. Typically in software compilers a schedule of operations is chosen according to a solution to the register allocation problem.

The exact solution to the register assignment problem requires solution to the vertex coloring problem for a conflict graph where the vertices correspond to variables and an edge indicates simultaneously live variables. The number of available colors corresponds to the number of available machine registers. It has been shown that this problem is NP-complete for general graphs[GJ79]. Hence heuristics solutions are commonly used. Most popular heuristics for code generation use a specific order of execution of successor nodes (e.g., left neighbour first) in order to reduce the size of S_d^{Π} [ASU86].

In contrast to the register assignment in conventional software compilers which perform simultaneous register assignment and operation linearization, we devise a two step approach to estimation of data storage and software delays resulting from additional memory operations.

1. **linearize operations**, that is, find a schedule of operations.
2. Estimate register/memory **data transfer operations**.

This two-step approach is taken to preserve the flexibility in operation scheduling which may be constrained by timing constraints not present in traditional software compilers.

Figure 39 illustrates the steps in estimation of software performance.

5.4.1 Operation linearization

Linearization of G refers to finding a complete order of operations in $V(G)$ that is topologically consistent with the partial order in G. This complete order corresponds to a schedule of operations on a single resource, that is, the processor. In the presence of timing constraints, the problem of linearization can be reduced to the problem of 'task sequencing of variable length tasks with release times and deadlines' which is shown to be NP-complete in the strong sense [GJ79]. It is also possible that there exists no linearization of operations that satisfies all timing constraints. Exact and heuristic ordering schemes under timing constraints are described in [KM92a].

We use a simplified version of the heuristic ordering in [KM92a] based on topological sorting of the vertices in the acyclic flow graphs. This sorting is performed based on a vertex elimination scheme that repetitively selects a zero in-degree vertex (i.e., a root vertex) and outputs it. The following procedure *linearize* outlines the algorithm. The input to the algorithm is a constraint graph model consisting of forward and backward edges as defined in Section 4.2 of Chapter 4. Recall, a backward edge represents a maximum delay constraint between the initiation times of two operations, whereas a forward edge represents a minimum delay constraint between the operation initiation times. By default, a non-zero operation delay leads to a minimum delay constraint between the operation and its immediate successors.

The algorithm consists of following three steps indicated by the symbol ($\triangleright$):

1. Select a root operation to add to the linearization,
2. Perform timing constraint analysis to determine if the addition of the selected root operation to the linearization constructed thus far leads to a feasible complete order, else select another root vertex,
3. Eliminate selected vertex and its dependencies, update the set of root operations.

The main part of the heuristic is in selection of a vertex to be output from among a number of zero in-degree vertices. This selection is based on the criterion that the induced serialization does not create a positive weight cycle in the constraint graph. Among the

```
linearize(G = (V, E_f ∪ E_b)) {
    δ(s) = 0; Q = { source vertex of G };          /* initialize */
    if positive_cycle(G)
        exit;                                      /* no valid linearization exists */
    repeat {
▷I      v = extract head(Q);                       /* vertex with smallest urgency label */
        G' = G;                                    /* construct new constraint graph */
        add edge (s, v) in G' with weight = δ(s);  /* select a candidate vertex */
        for all w ∈ Q and w ≠ v                    /* linearize candidates's siblings */
            add edges (v, w) in G' with weight = δ(v);
▷ II    if positive_cycle(G')                      /* not a feasible linearization */
            mark and move v to tail of Q;          /* discard candidate */
        else {                                     /* we have a good candidate */
▷ III       if v is head of an edge, (u, v) ∈ E_b
                δ(u, v) = δ(u, v) − δ(s);
            for all w ∈ succ(v) s.t. pred(w) = ∅
                Q = Q + {w};                       /* update Q with new root vertices */
            remove v from Q; output v;             /* delete vertex v and its successor- */
            δ(s) = δ(s) + δ(v);                    /* -edges in G' */
            G = G';
            sort Q by urgency labels;
        }
    } until Q = ∅;
}
```

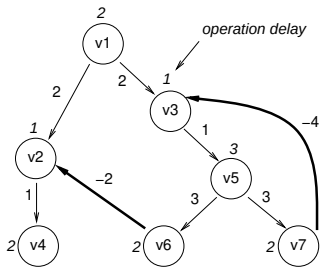
available zero in-degree vertices, we select a subset of vertices based on a two-part criteria. One criterion is that the selected vertex does not create any additional dependencies or does not modify weights on any of the existing dependencies in the constraint graph. For the second criterion, we associate a measure of *urgency* with each source operation and select the one with the least value of the urgency measure. This measure is derived from the intuition that a necessary condition for existence of a feasible linearization (i.e., scheduling with a single resource) is that the set of operations have a schedule under timing constraints *assuming unlimited resources*. A feasible schedule under no resource constraints corresponds to an assignment of operation start times according the lengths of the longest path to the operations from the source vertex. Since a program thread contains no $\mathcal{ND}$ operations, the length of this path can be computed. However, this path may contain cycles due to the backward edges created by the timing constraints. A feasible schedule under timing constraints is obtained by using the operation slacks to determine the longest path delays to operations. The length of the longest path is computed by applying an iterative algorithm such as Bellman-Ford algorithm or more efficiently Liao-Wong algorithm [LW83] that repetitively increases the path length until all timing constraints are met. This scheduling operation is indicated by the procedure *positive_cycles()* that either fails when it detects a positive cycle in the constraint graph or returns a feasible schedule. In case, if the algorithm fails to find a valid assignment of start times, the corresponding linearization also fails since the existence of a valid schedule under no resource constraints is a necessary condition for finding a schedule using a single resource. In case a feasible schedule exists, the operation start times under no resource constraints define the urgency of an operation.

The two criteria for vertex selection are applied in reverse order if a linearization fails. At any time, if a vertex being output creates a serialization not in the original flow graph, a corresponding edge is added in the constraint graph with weight equals delay of the previous output vertex. With this serialization, the constraint analysis is performed to check for positive cycles, and if none exists, the urgency measure for the remaining vertices is recomputed by assigning the new start times, else the algorithm terminates without finding a feasible linearization.

Since the condition for a feasible linerization used in the urgency measure is not

sufficient, therefore, the heuristic may fail to find any feasible linearization. It is also possible for this heuristic to fail while there may exist a valid ordering. Under such conditions an exact ordering search that considers all possible topological orders can be applied. The following example illustrates the linearization procedure.

Example 5.4.5. Operation linearization.



Consider the flow graph shown in the figure above.

Initialize: $Q = \{v_1\}$, $\delta(s) = 0$.

By applying the procedure *positive_cycle* on this graph, we get the following assignment of operation urgency labels, σ.

operation	v_1	v_2	v_3	v_4	v_5	v_6	v_7
label, $\sigma(v)$	0	4	2	5	3	6	6

Iteration = 1:

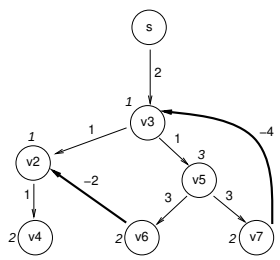
▷I: $v = v_1$. Add edge (s, v_1) with weight = 0.

▷II: no positive cycle. Feasible.

▷III: $Q = \{v_2, v_3\}$, **output =** v_1. $\delta(s) = 0 + \delta(v_1) = 2$. Since $\sigma(v_2) = 4$ and $\sigma(v_3) = 2$, Q is sorted to be $Q = \{v_3, v_2\}$ where the first element represents the head of Q.

Iteration = 2:

▷I: Candidate $v = v_3$. The new constraint graph G' is shown below:



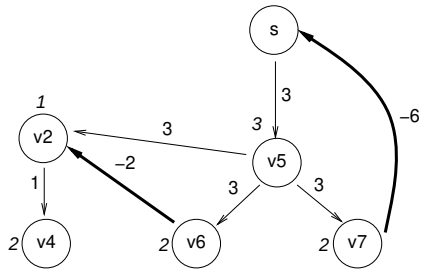
▷II: no positive cycle. Feasible. The assignment of the urgency labels:

operation	s	v_2	v_3	v_4	v_5	v_6	v_7
label, $\sigma(v)$	0	4	2	5	3	6	6

$\triangleright$III: $(v_7, v_3) \in E_b \Rightarrow \delta(v_7, v_3) = -4 - 2 = -6$. $Q = \{v_2, v_5\}$. **output = v_3**. $\delta(s) = 2 + \delta(v_3) = 3$. Urgency, $\sigma(v_2) = 4, \sigma(v_5) = 3$. Q is sorted as $Q = \{v_5, v_2\}$.

Iteration = 3:

$\triangleright$I: Candidate $v = v_5$. Add edge (s, v_5) with weight = $\delta(s) = 3$. The new constraint graph G' is shown below:



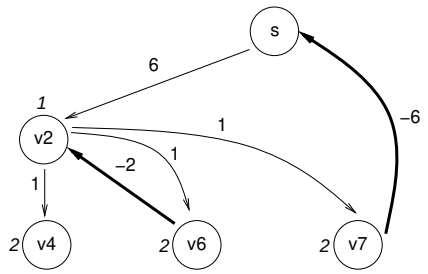
$\triangleright$II: no positive cycle. Feasible. The assignment of the urgency labels:

operation	s	v_2	v_4	v_5	v_6	v_7
label, $\sigma(v)$	0	6	7	3	6	6

$\triangleright$III: $Q = \{v_2, v_6, v_7\}$. **output = v_5**. $\delta(s) = 3 + \delta(v_5) = 6$. Urgency, $\sigma(v_2) = \sigma(v_6) = \sigma(v_7) = 6$.

Iteration = 4:

$\triangleright$I: Candidate $v = v_2$. Add edge (s, v_2) with weight = $\delta(s) = 6$. The new constraint graph G' is shown below:

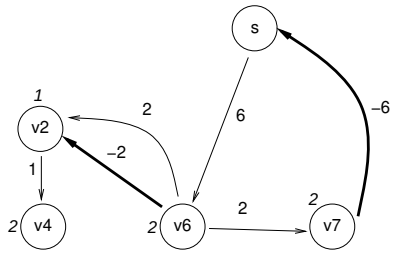


$\triangleright$II: positive cycle. Mark and move v_2 to tail of Q. $Q = \{v_6, v_7, v_2\}$.

Iteration = 5:

$\triangleright$I: Candidate $v = v_6$. The new constraint graph G' is shown below.

$\triangleright$II: positive cycle. $Q = \{v_7, v_2, v_6\}$.



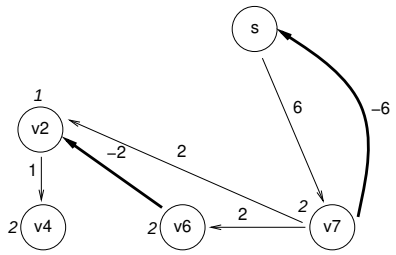
Iteration = 6:

▷I: Candidate $v = v_7$. The new constraint graph is shown below.

▷II: no positive cycle. Feasible. The assignment of the urgency labels:

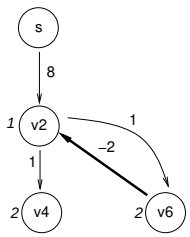
operation	s	v_2	v_4	v_6	v_7
label, $\sigma(v)$	0	8	9	8	6

▷III: $Q = \{v_2, v_6\}$. **output = v_7**. $\delta(s) = 6+2 = 8$. Urgency, $\sigma(v_2) = \sigma(v_6) = 8$.



Iteration = 7:

▷I: Candidate $v = v_2$. The new constraint graph is shown below:



▷II: no positive cycles. Feasible. The assignment of urgency labels: $\sigma(s) = 0, \sigma(v_2) = 8, \sigma(v_4) = 9, \sigma(v_6) = 9$.

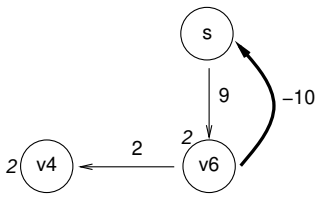
▷III: Since $(v_6, v_2) \in E_b \Rightarrow \delta(v_6, v_2) = -2 - 8 = -10$. **output = v_2**. $Q = \{v_6, v_4\}$.

Iteration = 8:

▷I: Candidate $v = v_6$. The new constraint graph is shown below:

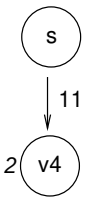
▷II: no positive cycles. Feasible. $\sigma(s) = 0, \sigma(v_6) = 10, \sigma(v_4) = 12$.

▷III: $Q = \{v_4\}$, $\delta(s) = 9 + 2 = 11$. **output = v_6**.



Iteration = 9:

▷I: Candidate $v = v_4$. The new constraint graph is shown below:



▷II: no positive cycles. Feasible.

▷III: $Q = \emptyset$. **output = v_4**.

Thus, the linearization returned by the algorithm is $v_1,\ v_3,\ v_5,\ v_7,\ v_2,\ v_6,\ v_4$. □

5.4.2 Estimation of register, memory operations

The number of read and write accesses is related to the amount and allocation of static storage, $S_d^{II}(G)$. Since it is difficult to determine actual register allocation and usage, some estimation rules are devised.

Let $G^D = (V,\ E^D)$ be the data-flow graph corresponding to a flow graph model, where every edge, $(v_i, v_j) \in E^D$ represents a data dependency, that is, $v_i \succ v_j$. Vertices with no predecessors are called source vertices and vertices with no successors are defined as sink vertices. Let $i(v), o(v)$ be the indegree and outdegree of vertex v. Let $n_i = |\{\text{source vertices}\}|$ and $n_o = |\{\text{sink vertices}\}|$. Let r_r and r_w be the number of register read and write operations respectively. Finally, recall that m_r represents the number of memory read operations and m_w represents the number of memory write operations.

Each data edge corresponds to a pair of read, write operations. These read and write operations can be either from memory (Load) or from already register-stored values. Register values are either a product of load from memory or a computed result. Clearly, all values that are not computed need to be loaded from memory at least once (contributing to m_r). Further, all computed values that are not used must be stored into the memory at least once (and thus contribute to m_w). Let R be the total number of unrestricted (i.e., general purpose) registers available (not including any registers needed for operand storage). In case the number registers R is limited, it may cause additional memory operations due to register spilling. A register spill causes a register value to be temporarily stored to and loaded from the memory. This spilling is fairly common in RM/MM processors and coupled with non-orthogonal instruction sets, result in a significant number of data transfers either to memory or to register operations (the latter being the most common). The actual number of spills can be determined exactly given a schedule of machine-level operations. Since this schedule is not under direct control, therefore, we concentrate on bounds on the size of the spill set, Ξ.

Case I: $R = 0$ In this case, for every instruction, the operands must be fetched from memory and its result must be stored back into the memory. Therefore,

$$m_r = |E| \tag{5.57}$$

$$m_w = |V| \tag{5.58}$$

Note that each register read results in a memory read operation and each register write results in a memory write operation, ($r_r = m_r$) and ($r_w = m_w$).

Case II: $R \geq R_l$ where R_l is the maximum number of live variables at any time. In this case no spill occurs as there is always a register available to store the result of every operation.

$$m_r = n_i \leq |V| \leq |E| \tag{5.59}$$

$$m_w = n_o \leq |V| \tag{5.60}$$

Case III: $R < R_l$ At some operation v_i there will not be a register available to write the output of v_i. This implies that some register holding the output of operation v_j will need to be stored into the memory. Depending upon the operation v_j chosen, there will be a register spill if output of v_j is still live, that is, it is needed after execution of operation v_i. Of course, in the absence of a spill, there will be no effect of register reallocation on memory read/write operations.

Let $S \subset V$ be the set of operations that are chosen for spill.

$$m_r = n_i + \sum_S o(v_i) \leq \sum_V o(v_i) = |E| \quad (5.61)$$

$$m_w = n_o + |S| \leq |V| \quad (5.62)$$

Clearly, the choice of the spill set determines the actual number of memory read and write operations needed. In software compilation, the optimization problem is then to choose a spill set, S such that $\sum_S o_(v)$ is minimized. This is another way of stating the familiar register allocation problem. As mentioned earlier, the notion of liveness of an output $o(v)$ of an operation, v can be abstracted into a *conflict graph*.

The optimum coloring of this graph would provide a solution to the optimum spill set problem. This problem is shown to be NP-complete for general graphs [Cha82]. In case of a linearized graph with no conditionals, nodes in the corresponding conflict graph correspond to intervals of time and an edge indicates an overlap between two intervals. Therefore, the conflict graph is an interval graph. For linearization purposes, operations on conditional paths are treated as separate subgraphs that are linearized separately. Recall that no timing constraint are supported on operations that belong to separate conditional paths. For interval graphs, the coloring problem can be solved in polynomial time. That is, a coloring of vertices can be obtained that uses the minimum number of colors, for example, by using the left-edge algorithm [HS71]. However, a problem occurs when the number of registers available is less than the minimum number of registers needed. In this case, outputs from a set of vertices should be spilled to memory and the conflict graph modified accordingly so that the new conflict graph is colorable. We use the following heuristic to select operations for the spill. First, a conflict graph, G_I for a given schedule is built by drawing an edge between v_i and v_j if any of the output edges of v_i span across

v_j. From this conflict graph, we select a vertex, v with outdegree less than R. This vertex is then assigned a register different from its neighbours. From this we construct a new conflict graph G'_I by removing v and its fanout edges from G_I. The procedure is then repeated on G'_I until we have a vertex with outdegree greater than or equal to R. In this case, a vertex is chosen for spilling and the process is continued. Example 5.4.6 illustrates the procedure.

For these calculations, we assume that each v is implemented as a closed sequence of assembly language instructions, though it is possible that the source language compiler may rearrange operations. The effect of this rearrangement, however, can assumed to be only to reduce the total register usage requirements.

Effect of multiregister nodes

The register usage of compilers is determined by the generation of *rvalue* and *lvalue* [ASU86]. The rvalue is the result or value of evaluation of an expression (or simply the right-hand side of an assignment). In the case of logic nodes, the (recursive) organization of equations gives an estimate of the number of rvalues needed (plus additional rvalues due to shifts). For most assignment statements, the left side generates a lvalue and the right side generates a rvalue. However, in case of pointer assignments and structure and indirect member references (common in logic nodes) left hand side also generates rvalues which are subsequently assigned to appropriate lvalue also generated by the left side. This generation of an rvalue by the left side happens in two cases:

1. Write operation

2. Logic operations

In both cases, the left hand side generates a rvalue that is assigned to a left hand generated lvalue.

We extend our estimation procedure for the spill set in the case of operation vertices with multiple register usage simply by using a weighted graph model where the weight of each vertex is given by the number of rvalues it generates, that is, $G^D = (V,\ E,\ \omega)$ where $\omega(v) = |rvalue(v)|$ and $\omega(e = v_i \succ v_j) = \omega(v_i)$. The above relations hold by

replacing

$$|V| = \sum_{V} \omega(v) \tag{5.63}$$

$$|E| = \sum_{E} \omega(e) \tag{5.64}$$

The following procedure *single_thread_static_storage(G)* determines maximum number of live variables, R_l, in a linearized graph model, G. We assume that each operation vertex requires at least one cycle and hence any data transfer across operation vertices in the flow graph requires a *holding* register.

Input: flow graph model, $G(V, E)$
Output: $S(G)$, static storage for a linear code implementation of G
single_thread_static_storage(G) {
  H = *linearize*(G) /* linearize vertices */
  count = storage = 0 ;
  $\forall\, u \in V(H)$ { /* determine max live variables */
    $\forall\, v \in$ *succ*(u)
      count = count + $\omega(u > v)$; /* add new registers */
    $\forall\, v \in$ *pred*(u)
      count = count - $\omega(v > u)$; /* subtract registers for completed operations */
    storage = *max*(count, storage) ;
  }
  return storage
}

RM ISA Architectures

Practical RM ISA architectures feature small register sets with non-orthogonal instructions. That makes register spills a very common occurrence in the compiled code. But more importantly, due to non-orthogonality of instructions, a substantial number (up to 27 %) of *inter-register* data transfer instructions are generated to position data to appropriate registers from other registers [HP90]. These instructions do not affect the data storage, S_d^{Π}, but these alter the size of the program thread and its delay. It is hard to model such instructions since these are dependent upon actual algorithms used in software compilation. A better technique would be to perform compilation of the assembly code within

the system. This provides a greater handle over the estimation of generated instructions. The following conjecture is suggested to estimate the additional operations.

Conjecture 5.1 *For a given graph model, $G = (V, E)$, the following sum*

$$\Gamma = m_r + m_w + r_m = |V| + |E| \tag{5.65}$$

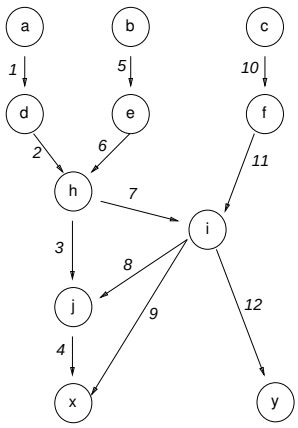
is constant for all architectures. Here r_m represents the number of inter-register transfer operations.

The intuitive reasoning behind this conjecture is that for a machine with no general purpose registers, there will be no need for inter-register operations since operands can be loaded directly into the required operand registers, and thus $r_m = 0$. This corresponds to the case I discussed earlier. In the other two cases, the total number of data movement operations can not be worse than the case with no registers.

Based on this conjecture, we can estimate the inter-register transfer operations by first computing $m_r + m_w = f(R)$ as a function of the number of available registers, R and then applying Equation 5.65 above.

Example 5.4.6. Linearization and data transfer operations.

Consider the flow graph, G, shown in figure below consisting of 11 vertices and 12 edges with $n_i = 3$ and $n_o = 2$.

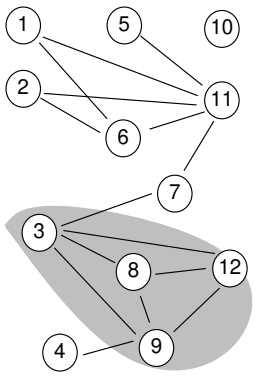


Each vertex produces an output variable that is named by the vertex label. Vertices a, b, c are input read operations, and vertices x, y are output write operations. A depth-based linearization results in the following order

$$c, f, b, e, a, d, h, i, y, j, x$$

R_l	*Spill set,* Ξ	*Memory operations,* $m_r + m_w$
≥ 4	$\{\}$	5
3	$\{9\}$	7
2	$\{9,\ 3,\ 11\}$	11

which gives the maximum number of live variables, $R_l = 4$ according to algorithm *single_thread_storage*. R_l is the size of the largest clique in the interval graph shown below. In the interval graph, G_i, vertices represent edges of G and an edge between



two vertices in G_i indicates overlap between the corresponding edges in G.

For $R \geq R_l = 4$: register assignment can be done in polynomial time. The spill set, $\Xi = \emptyset$. Therefore, $m_r = n_i = 3$ and $m_w = n_o = 2$. Total number of memory operations = 5.

For $R = 3$ the largest clique in G_i should be of cardinality 3 or less. For $\Xi = \{9\}$, the total number of memory operations = 7. For $\Xi = \{9, 3, 11\}$, the maximum number of live registers is reduced to 2, while the number of memory operations increases to 11. Note that in the worst case of static storage assignment for all variables in G, there would be $11 + 12 = 23$ memory operations. □

Note that we are not directly trying to minimize total register usage by the object program since that abstraction level belongs to the software compiler. The objective of spill set enumeration is to arrive at an estimate of memory operations assuming that the program is compiled by a reasonably sophisticated software compiler that achieves optimum register allocation when the maximum number live variables is less than the

number of available registers. Clearly this is only an estimate since the actual register allocation will vary from compiler to compiler. In order to get insight into the effect of software compilers, we briefly mention the common optimizations performed by most software compilers in the following section. This discussion is not directly relevant to the hardware-software co-synthesis scheme proposed by this thesis and for more detailed discussions on the subject the reader is referred to [ASU86] [HP90].

5.4.3 Compiler effects

Compiler directed optimizations fall into following categories:

- Source level optimizations, for example, like procedure inlining.
- Basic-block level optimizations: common subexpression elimination (CSE), constant propagation, expression tree height reduction.

  Structure preserving transformations: dead-code elimination; renaming of temporary variables; interchange of two independent adjacent statements.

  Note that these two categories of transformations are already taken care of by the HDL compiler.
- Global optimization: copy propagation, code motion, induction variable elimination (and other loop related optimizations)
- Machine register allocation
- Machine related optimizations: operator strength reduction, pipeline scheduling, branch offset minimization

Of these, machine register allocation accounts for perhaps the most increase in efficiency (20-50 %). Register allocation is possible and most effective for local variables which are conventionally stored in the runtime stack. These locals are defined by the storage $M(G)$ associated with the flow graph model.

5.4.4 Software data size and performance tradeoffs

There exists a tradeoff between code size and data size for a given graph implementation into software. The storage used by G can be packed to yield smaller data size, but this results in a larger code size and therefore, greater execution times. On the other hand, unpacked storage leads to larger data size but the code size is small. The key problem, therefore, is to choose variables to be packed. One heuristic is to make a distinction between data and control variables by treating variables that are assigned and tested to be control variables. An alternative is to use a memory cost function, $\mathcal{C}$, that assigns a cost value for a given memory size. This cost function is based on a model of storage chips which come in various sizes m_1 to m_k (e.g., 4K, 16K, 256K, 1M, 4M, 16M) with cost per chip of c_1 to c_k.

$$\mathcal{C}(M(G)) = \left\lceil \frac{M}{m_i} \right\rceil \cdot c_i \qquad m_i \leq M < m_{i+1} \tag{5.66}$$

Based on this cost model for the memory storage, we can determine the cost of the minimum possible data storage based on packing of all the variable. The minimum data storage is given by:

$$M_{min} = \left\lceil \frac{\sum_1^N b_i}{W} \right\rceil \tag{5.67}$$

where W is the width of the memory. Similarly, we can determine the maximum data storage based on no packing, that is, every data is aligned to a word boundary:

$$M_{max} = \sum \left\lceil \frac{b_i}{W} \right\rceil \tag{5.68}$$

The size of the software is given as the sum of the data size and the program size. In case of a packed data, the program size increases somewhat due to additional operations needed to access a packed data value. Between these two extremes of data storage, a set of variables to be packed can be selected that leads to a lower cost of the memory.

5.5 Software Synthesis

The task of synthesis of software from flow graphs is divided into the following four steps as shown in Figure 40:

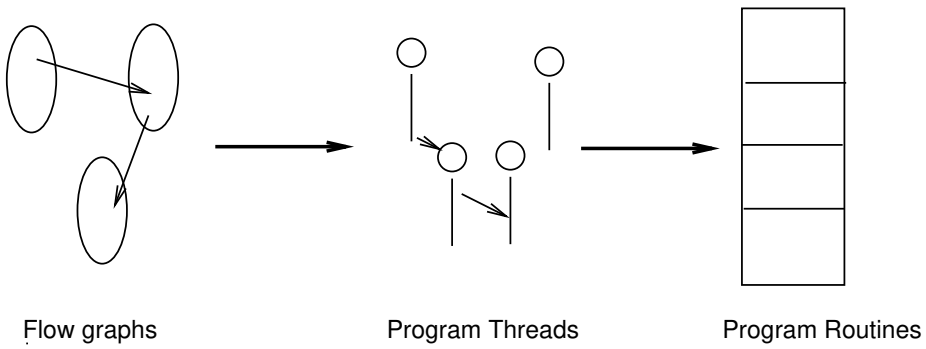


Figure 40: *Steps in generation of the software component*

Step 1: Generation of linearized sets of operations or *program threads* **from flow graphs**. The operations that belong to a program thread are identified by system partitioning (described in Chapter 6). The task of program thread generation to ensure correct dependencies between program threads based on operation dependencies. An optimization called *convexity serialization* is applied to reduce the overhead due to these dependencies. Convexity serialization is followed by linearization of operations in subgraphs of the flow graph models. Program thread generation is discussed in Section 5.6.

Step 2: Generation of program routines from program threads. In addition to operations in the program threads, a program routine also contains operations that make it possible to achieve concurrency and synchronization between program threads. The essential problem in program routine generation is implementation of various program threads for execution on a processor that supports only a single thread of control at any time. This issue is discussed in Section 5.7.

Step 3: Code generation from program routines. As mentioned earlier, for purposes of retargetability, we generate C-code from program routines. Code generation requires translation of operations defined in the flow graph model into corresponding operations in C, a high-level programming language, identification of memory locations, binding of variables to memory addresses. This is discussed in Section 5.8.

Step 4: Compilation of program routines into processor assembly and object code. C-programs are compiled using existing software compiler for the target processors. Some issues related to the interface of the object code to the underlying processor and ASIC hardware must be resolved at this level. These are discussed in Section 5.9.

5.6 Step I: Generation of Program Threads

A program thread refers to a linearized set of operations in the flow graph model. Operations in a program thread are identified as a result of system partitioning discussed in next chapter. By construction, a program thread is initiated by a synchronization operation, such as a blocking communication or a loop synchronization operation. However, within each thread all operations have a fixed delay. The (unknown) delay in executing the synchronization operation appears as a delay in scheduling the program thread and it is not considered a part of the thread latency. Therefore, for a given re-programmable device the latency of each thread is known statically.

Recall that the *wait* operation is referred as a synchronization operation. Depending upon the number of synchronization operations in a flow graph, the graph model can be implemented as a single or multiple program threads. In the absence of any synchronization, a simple graph model can be translated into a single program thread by ordering all the operations of the graph model (assuming that such an order exists under timing constraints).[3] On the other hand, a hierarchical system model is implemented as a set of program threads where each thread corresponds to a graph in the model hierarchy.

A partitioning of the system model results in identification of subgraphs in a flow

[3]Note that if no linearization of a graph model without $\mathcal{ND}$ operations exists under timing constraints, then there does not exist any linearization using multiple program threads that would meet the timing constraints. This is due to the fact that the constraint graph model can be partitioned into strongly-connected-components (SCCs), such that a linearization of the graph model exists if and only if linearization for each SCC exists [Ku91]. Since the SCC contains no synchronization operations, therefore, any implementation into multiple threads will only be worse than a single thread implementation due to additional runtime overheads required.

graph model that belong to separate program threads. There may be dependencies between operations that belong to separate subgraphs and hence separate threads. Representation and incorporation of these dependencies is discussed next.

Control Flow in the Software Component

Since multiple program threads may be created out of a graph model each starting with an $\mathcal{ND}$ operation, software synchronization is needed to ensure correct ordering of operations within the program threads and between different threads. Some threads may be hierarchically related, that is, dependent whereas some program threads may need to be executed concurrently based on the concurrency among the corresponding graph models. Concurrency between program threads can be achieved by using an inter-leaved computation model as explained later in this section.

Example 5.6.7. Concurrent and dependent program threads.

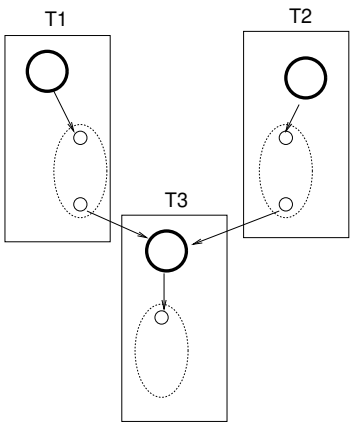


Figure above shows subgraphs and corresponding program threads, T_1, T_2 and T_3. Bold circles indicate synchronization operations. Threads T_1 and T_2 are concurrent where as thread T_3 is dependent upon T_1 and T_2. □

Since the total number program threads and their dependencies are known statically, the programs threads are constructed to observe these dependencies. A program thread is in one of the following three states *detached*, *enabled* or *running*. While the details of the mechanism to enable and select from a set of program threads are described in Chapter 7, it is sufficient to note here that a detached thread is first enabled before it is

able to run. The dependencies between program threads are, therefore, implemented by altering the enabling condition for program threads. In our implementation of hardware-software systems, this enabling of program threads is achieved by a special run-time first-in/first-out (FIFO) structure, called **control FIFO**. A thread is enabled only when its located in the control FIFO. Before detaching, a thread performs one or more *enqueue* operations to this FIFO for its dependent threads as shown in Example 5.6.8 below.

Example 5.6.8. Inter-thread control dependencies.

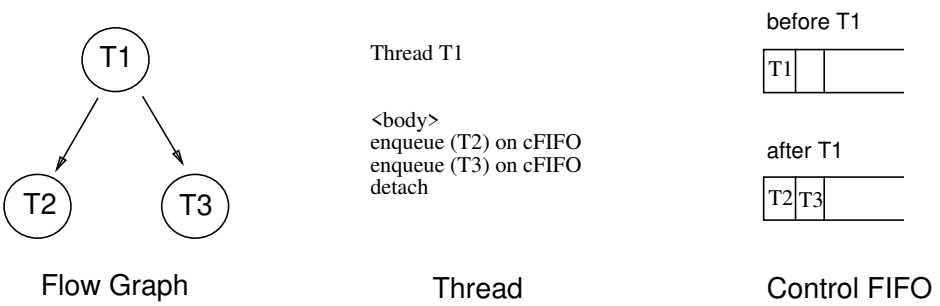


Figure 41: *Use of enabling condition to build inter-thread dependencies.*

Figure 41 shows program threads T_2 and T_3 that are enabled by the program thread T_1. The `<body>` refers to the (linearized) set of operations from the corresponding graph models for program thread T_1. Control dependency from thread T1 to T2 is built into the code of T1 by the enqueue operation on the control FIFO. □

A thread dependency on more than one predecessor thread (that is a multiple indegree (fanin) node in the flow graph) is observed by ensuring multiple enqueue operations for the thread by means of a counter. For example, a thread node with a indegree of 2 would contain a synchronization preamble code as indicated by the while statement shown in Example 5.6.9 below.

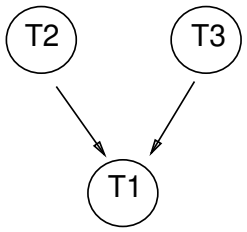
Example 5.6.9. Thread with multiple input control dependencies.

Here thread T_1 is enabled by threads T_2 and T_3. This is accomplished by two enqueue operations by T_2 and T_3 before the thread T_1 is ready to run. □

Control transfer for multiple fanin nodes entails program overheads that add to the latency of the corresponding threads. For this reason, an attempt should be made to reduce

```
Thread T1

while (count != 1)
{
        count = count + 1;
        detach
}
<body>
count = 0
enqueue <successor threads> on cFIFO
detach
```



multiple dependencies for a program thread through a careful dependency analysis. In case of multiple outdegree nodes in the flow graph, a necessary serialization among enabling of successor threads occurs. However, this serialization is of little significance since there exists only a single re-programmable component.

Convexity serialization

In case there is a dependency between two program threads caused by a dependency between two operations within the bodies of the program threads, this leads to an enabling operation for the successor thread within the body of the enabling thread. The dependent thread is run to its execution until just before the dependent operation and then it is detached before executing the dependent operation. This leads to overheads in execution of dependent program threads that must be resumed by the runtime scheduler before completion. An alternative is to modify these 'in-body' dependencies between program threads by making the subgraphs corresponding to program threads convex.

We define a subgraph to be *convex* if the subgraph has a single entry and exit operations. For a convex subgraph, the corresponding program thread once invoked can run its execution to completion without any need to detach in order to observe the dependencies. A convexity serialization is based on the property of partial orders that an edge from operation u to v indicating dependency, $u > v$ can be sufficiently replaced by either a dependency $u > w$ where w is a (transitive) predecessor of v, that is, $w >^* v$; or by a dependency $w > v$ where w is a (transitive) successor of u, that is, $u >^* w$. Convex serialization is used to obtain convex subgraph. This results in a potential loss of concurrency, and timing constraint analysis must be performed on the modified constraint graphs to ensure constraint satisfiability is maintained. However, it makes the task of

routine implementation easier since all the routines can be implemented as independent programs with statically embedded control dependencies.

Example 5.6.10. Convex subgraphs.

Figure 42 below shows a flow graph that is to be implemented in software. Operations 'b' and 'c' represent synchronization operations (defined as anchors in Chapter 4. Recall that the anchor set of an operation refers the set of $\mathcal{ND}$ operations that the operation (transitively) depends upon. The determination of which thread an operation belongs to is made by examination of its anchor set. Operations with the same anchor set belong to the same program thread. In this example, the shaded vertices share the same anchor set containing operations 'b' and 'c'.

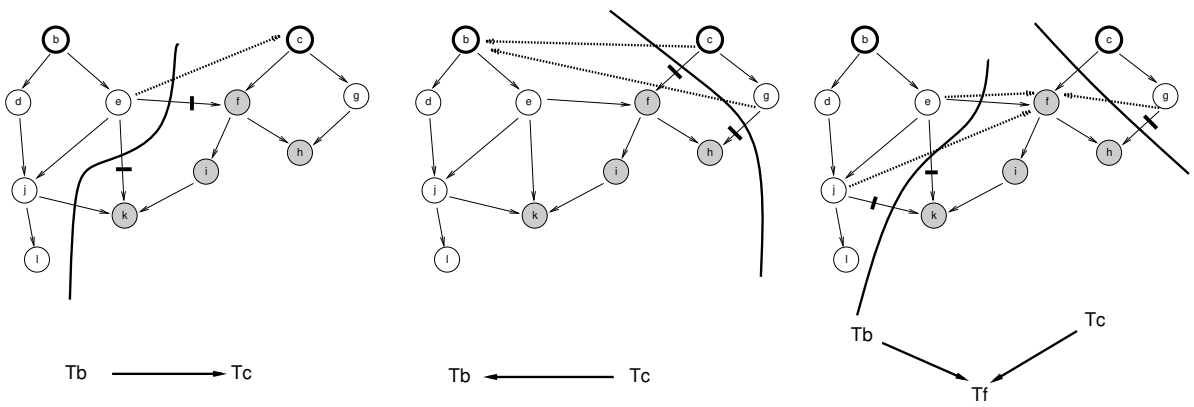


Figure 42: *Convexity serializations and possible thread implementations.*

There are three possible ways to implement the program threads as shown. In all three cases the dependencies created due to convexity serialization are shown by dashed arrows. Let consider the first case that leads to two program threads, T_b and T_c. The edge (e, f) can be replaced by an edge (e, c) since c is a predecessor of f. Similarly, edge (e, k) can be replaced by (e, c). This convexity serialization has two effects: one, it creates two dependent program threads, where T_c depends upon T_b and secondly, the edge (e, c) creates additional dependency (e, g) that was not present in the original flow graph. This leads to a potential loss of concurrency mentioned earlier. □

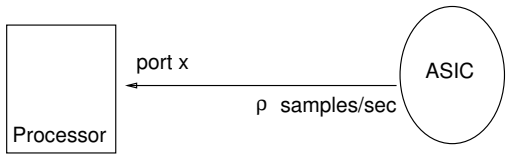
Though only a feature of representation, the use of hierarchy to represent control flow is well suited to eventual implementation of the software component as a set of program routines. Since all the operations in a given graph model are eventually executed, the corresponding routines can be constructed with known and fixed latencies as explained

earlier. As with the graph model, the uncertainty due to data-dependent loop operations is related to invocations of the individual routines corresponding to the loop body.

Rate constraints and loop implementations

A software implementation consisting of dynamic invocations of fixed latency program threads simplifies the task of software characterization for satisfaction of data rate constraints. Satisfaction of imposed data rate constraints depends on the performance of the software component. It was shown in previous chapter that minimum rate constraints on flow graphs with $\mathcal{ND}$ operations lead to $\mathcal{ND}$-cycles which can be checked for marginal satisfiability of the rate constraints. The resulting producer consumer system can be transformed by means of buffers such that it is better able to meet the rate constraints. In the following we describe the software realization of these transformations. The simplest case occurs for *weakly preindexed* loop operations, for which the loop index is computed before invocation of the loop and is unaffected by the body of the loop operation. For these loops, the *dynamic* loop execution model can be transformed into a *pseudo-static* loop execution model as follows. Consider, for example, a software component that consists of reading a value followed by an $\mathcal{ND}$ loop operation shown in Example below.

Example 5.6.11. Consider a mixed implementation shown by the figure below.



The ASIC component sends to the processor some data on port x at an input rate constraint of ρ samples/sec. The function to be implemented by the processor is modeled by the following HDL process fragment.

```
process test(x, ...)
   in port x [SIZE];
{
   ...
   read x ;
   repeat
   {
      <loop-body>
      x = x - 1 ;
   } until (x <= 0);
}
```

```
Thread T1
read
enqueue T2
detach
```

```
Thread T2
<loop_body>
x = x - 1
if not (x <= 0)
    enqueue T2
detach
```

x is a boolean array that represents an integer. In its software implementation, this behavior is translated into a set of two program threads shown on the right, where one thread performs the reading operations, and the other thread consists of operations in the body of the loop. For each execution of thread T1 there are x execution of thread T2. □

Due to $\mathcal{ND}$ loop operation, the input data rate at port x is variable and is dependent upon value of x as a function of time. For each invocation of thread T1 there are x invocations of thread T2. In other words, thread T1 can be resumed after x invocations of thread T2. In absence of any other data-dependency to operations in the loop body, thread T1 can be restarted before completing all invocations of thread T2 by *buffering* the data transfer from thread T1 to T2. Further, if variable x is used only for indexing the loop iterations (that is, weakly preindexed loops), the need for inter-thread buffering can be obviated by accumulating value of x into a separate loop counter as shown in example below. We call such an implementation of a loop construct in software a *pseudo-static* loop based on the fact that an upper bound on the number of iterations of the loop body is statically determined by the data rate constraints on inputs and outputs that are affected by the $\mathcal{ND}$ loop operation.

Example 5.6.12. Transformation of the $\mathcal{ND}$ loop in Example 5.6.11 into a pseudo-static loop

```
process test(x, ...)                    |     Thread T1    Thread T2
   in port x [SIZE]                     |
{                                       |     read         <loop-body>
   integer repeat-count = 0 ;           |     add op       repeat-count--
   read x ;                             |     enqueue T2   if !(repeat-count <= 0)
   repeat-count = repeat-count + x ;    |     detach            enqueue T2
   repeat                               |                       detach
   {                                    |
      <loop-body>                       |
      repeat-count = repeat-count-1;    |
   } until (repeat-count <=0);          |
 }
```

For each execution of thread T1 there are $\max(x, m)$ execution of thread T2 where constant m is determined by input data rate constraint, ρ, on the read operation in T1 given by the relation: $\frac{1}{\rho} = (\lambda_{T1} + m \cdot \lambda_{T2}) \cdot \tau$ where the thread latencies λ_{T1} and λ_{T2} include synchronization overheads ($\overline{\gamma}$). τ denotes cycle time of the processor. □

In this case, we can provide a bound on the rate at which port is read by ensuring that the read thread, T1, is scheduled, say after utmost m iterations of the loop body.

Due to accumulation of repeat-count additional care must be taken to avoid any potential overflow of this counter. [Generally, overflow can be avoided if m is greater than or equal to the average value of x. In the extreme, it can be guaranteed not to overflow if m is at least maximum of x which is equivalent to assigning worst-case delay to the loop operation].

5.6.1 Implementation of inter-thread buffers

With concurrent program threads, to a certain extent, we can insulate the input/output data rates from variable delays due to other threads by buffering the data transfers between threads. Thus, the **inter-thread buffers** hold the data in order to facilitate multiple executions among program threads. Threads containing specific input/output operations are scheduled at fixed intervals via processor *interrupts* as shown in the Example 5.6.13 below. In this scheme, finite-sized buffers are allocated for each data-transfer between program threads. In order to ensure the input/output data rates for each thread, we associate a timer with every I/O operation that interrupts the processor once the timer expires. The associated interrupt service routine performs the respective I/O operation and restarts the timer. In case a data item is not ready the processor can send the previous output and (optionally) raise an error flag.

Example 5.6.13.

```
Thread T2           Timer Process            T1 (interrupt service routine)

<loop_body>         timer-- per clock tick   read x
x = x - 1           if (timer == 0)          load timer = CONSTANT
if not (x <=0)                 interrupt     enqueue (x) on dFIFO
   enqueue T2                                enqueue T2
else                                         detach
   x = dequeue dFIFO
detach
```

Thread T1 is now implemented into an interrupt service routine that is invoked at each expiration of the timer process. The timer process represents a processor timer (or an external hardware timer) that is used to generate interrupts at regular intervals. The interruption interval `CONSTANT` is determined by the rate constraint and latencies of interrupt service routines. dFIFO in the interrupt service routine refers to the buffer between threads T1 and T2. □

This scheme is particularly helpful in the case of widely non-uniform rates of production and consumption. In this case, the data transfer from processor to ASICs is handled by the interrupt routines, thereby leading to a relatively smaller program size for the cost of increased latencies of the interrupt service routines. Chapter 8 presents implementation costs and performance of this scheme.

Next, we consider the problem of software synchronization and scheduling mechanisms to make a hardware-software system design feasible.

5.7 Step II: Generation of Program Routines

Since the processor is completely dedicated to the implementation of the system model and all the program threads are known statically, the final program can be generated in one of following two ways.

1. Generate a single program routine that incorporates all the program threads. This approach is discussed in Section 5.7.2.

2. Provide for multiple-thread executions by means of operation interleaving as discussed in Section 5.7.1.

In the first case, we attempt to merge different routines and schedule all the operations in a single routine. The unbounded delay operations are explicitly handled either by busy-waiting or by executing specific context-switch operations. In the second case, concurrency between threads is achieved by interleaved execution on a single processor. In principle, operation interleaving can be as "finer-grained" as the primitive operations performed by the processor, that is the assembly instructions. However, we make a further assumption that interleaving is performed at the level of operations used in the flow graph model. This assumption is made to avoid otherwise excessive overheads due to implementation of concurrency at processor instruction level which is out of the scope of this work.

5.7.1 Concurrency in software through Interleaving: Coroutines

The problem of concurrent multi-thread implementation is well known[AS83]. In general, multiple program threads may be implemented as subroutines operating under a global task scheduler. However, subroutine calling adds overheads[4] which can be reduced by putting all the program fragments at the same level of execution. Such an alternative is provided by implementing different threads as coroutines [Con63]. In this case, the routines maintain a *co-operative* rather than a *hierarchical* relationship by keeping all individual data as local storage. The coroutines maintain a local state and willingly relinquish control of the processor at exception conditions which may be caused by unavailability of data (for example, a data dependency on another thread) or an interrupt. In case of such exceptions the coroutine switch picks up the processes according to a predefined priority list. Upon resumption a coroutine execution starts execution from the position where its was detached last. Implementation of the code for such a scheduler for coroutines takes approximately 100 bytes in an instruction set that supports both register and memory operands. To implement a coroutine in a general purpose microprocessor, each coroutine must have its own stack. There can be two approaches to make the context switch. One, each process in software knows which process will be executed next, so it may transfer directly to the next process. In this case, the coroutine transfer is executed in 34 instructions or 364 clock cycles (for the 8086). The second approach assumes a third process called a *task manager* whose function is to provide some priority scheme to the execution of the processes. This transfer in this case is more expensive, taking 728 clock cycles plus the task manager execution time.

5.7.2 Software implementation using description by cases

Any sequential program can be thought of as a finite-state machine with program counter acting as a state variable. Based on this concept, we can merge different routines and describe all operations in a single routine using the method of *description by cases* [Kin67]. This scheme is simpler than the coroutine scheme presented above. Here we construct a single program which has a unique state assignment for each synchronization

[4]The overheads are mostly due to operations to carry out runtime storage management.

Implementation	*Processor type*	*Overhead cycles*
Subroutine	R/M	728
Coroutine	R/M	364
Restricted Coroutine	R/M	103
Description by cases	R/M	85
Restricted Coroutine	L/S	19
Description by cases	L/S	35

Table 6: *Comparison of program thread implementation schemes*

operation. A global state register is used to store the state of execution of a thread. Transitions between states are determined by the runtime scheduling of different $\mathcal{ND}$ operations based on the data received.

This method is restrictive since it precludes use of nested routines and requires description as a single switch statement, which in cases of particularly large software descriptions, may be too cumbersome. Overhead due to state save and restore amounts to 85 clock cycles for every point of non-determinism when implemented on a 8086 processor. Consequently, this scheme entails smaller overheads when compared to the general coroutine scheme described earlier.

Table 6 summarizes program overhead for different implementation schemes. The processors are categorized based on their instruction set architectures as described in Section 5.1. *Overhead cycles* refers to the overhead (in cycles) incurred due each transfer operation from one program thread to another. A *Subroutine* implementation refers to translation of program threads to program subroutines that operate under a global task scheduler (or the *main* program). A *Coroutine* implementation reduces the overhead by placing routines in a co-operative, rather than hierarchical, relationship to each other. A *Restricted coroutine* implementation reduces the overhead further by suitably partitioning the on-chip register storage between program threads such that program counter is the only register that is saved/restored during a thread transfer. In case of R/M processors the case description scheme reduces the overhead by reducing amount of ALU operations in favor of a slight increase in memory input-output operations. By default, Vulcan generates coroutine based implementation of software.

5.8 Step III: Code Synthesis

This section describes the translation of program threads into corresponding C-code. Since, a program thread essentially consists of a sequence of assignments. Therefore, its translation into a corresponding imperative language like C can be performed as a syntax-directed translation of the program threads into C source-code. The *lvalue* for each assignment can be either a variable, or a bit-vector with a range. The *rvalue* of an assignment is an expression which either a variable or one or two operands with a unary or binary operation respectively.

This translation procedure is described using extended BNF notation. The following describes the terminal, non-terminal symbols and the printing actions associated with the syntax-directed translations of the program threads.

```
Terminal Symbols:
        VAR     = symbol
        SVAR    = symbol[h:l]
        CONST   = value
    UOP     =  ~ | -
    BIOP    = + | - | * | / | @ | >> | << | & | && | | | || | ^
print_assignment()
{
assignment: lvalue = rvalue

lvalue  : VAR
            print symbol = print_rvalue()
        | SVAR
    print symbol &= CLEAR_MASK
            print symbol |= print_rvalue() << l

}
print_rvalue()
{
rvalue  : opnd
        | UOP {print UOP} opnd
        | opnd BIOP {print BIOP} opnd
| FUNCTION {print name(} opnd opnd ... {print ) }

opnd    : VAR
            print symbol
        | SVAR
            print symbol >> l & ~(~0 << h-l+1)
        | CONST
            print value
}
```

5.9 Issue in Code Synthesis from Program Routines

As mentioned earlier, we generate C-code from partitioned graph models. The use of a high-level programming language for software generation provides the ability to generate

the corresponding object code for most commonly used processors. While this *retargetability* can be realized for most of the software component, there are certain program implementation issues that must be addressed while compiling and loading the generated C-programs. In this section, we address the major practical implementation issues.

5.9.1 Memory allocation

The C-compiler uses two kinds of memory structures: a *stack* for storing local variables in order to facilitate subroutine calls; and a *heap* for dynamically allocating memory space to run-time generated data structures. When using target systems with limited available memory (especially in case of microcontrollers where the on-chip memory is severely constrained), the unconstrained use of stack and heap space may lead to runtime exceptions that may make the software component non-functional. Fortunately, the use of both stack and heap can be avoided by performing static memory allocation in the generated program. Static memory allocation makes the generated program **non-recursive** and **non-reentrant**. The non-recursive nature of the software component is not an issue since the input graph models are themselves non-recursive, thus ruling out the possibility of recursion in the generated programs. A non-reentrant program cannot be entered by more than one task. This is usually a problem in case of general-purpose computing systems where a program execution must co-exist with other programs and the operating system software. In our application, the only restriction placed by non-reentrant code is that the main program and the interrupt service routines must not share any procedure calls.

5.9.2 Data types

The standard C programming languages supports the following data types: char, short int, int, long int, float and double. Format compatibility for the encoded/interpreted data types (types other than bit-vectors) becomes an issue when interfacing a general-purpose processor to external hardware such as A/D converters. Further, most standard C-compilers support declaration prefixes *const* and *volatile*. A *const*-declared data set

can be mapped to on-chip read-only memory (ROM). For variables declared as shared-storage between program threads and as memory-mapped I/O variables, the use of a *volatile* declaration preserves these variables from any compiler-driven optimizations.

5.9.3 The C Standard Library

The standard C-library contains procedures that are called by most C-programs. Most of these procedures are coded as C-programs thus making them portable across systems. However, some of these procedures are written as assembly programs. Commonly used assembly routines are *getchar()* and *putchar()* that are used for most I/O operations. These routines must be written for the target processor.

5.9.4 Linking and loading compiled C-programs

When using the routines from the standard C-library, only the routines used by the program are loaded into the object image. The object image consists of memory-relocatable modules. A hardware-software interface often contains fixed memory locations for interface semaphores, hardware devices addresses et cetera. When using relocatable object code, fixed addresses can be generated and used by the program by creating special relocatable modules that are loaded at fixed addresses during executions. Use of smaller relocatable modules for fixed-address generation avoids the problem of having to create fixed-address object modules for the entire software component. Example 5.9.14 shows how such modules can be used to address a fixed location interrupt vector table.

Example 5.9.14. Using relocatable modules to generate fixed-address locations

The interrupt-vector table is located at a fixed address 0xffd6. The following relocatable module `vector-table` contains pointers to various service routines. `vector-table` is compiled separately and loaded at address 0xffd6.

```
extern void reset();
extern void sci();
extern void spi();
...
void (* const vector-table[])() = {
   sci(),                          /* SCI service routine */
   spi(),
   ...
   reset,
   };
```

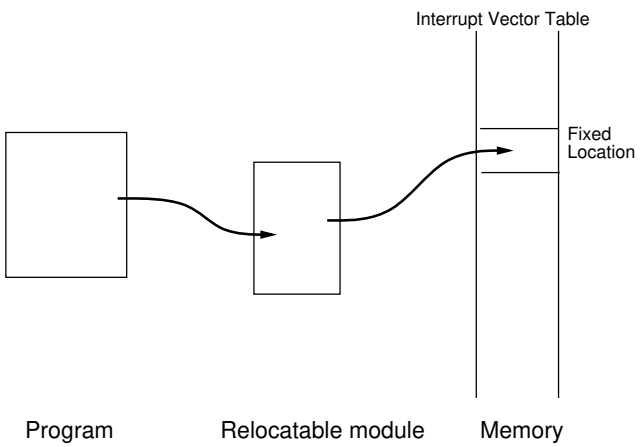


Figure 43: *Generating fixed addresses from C-programs*

□

5.9.5 Interface to assembly routines

For a variety of reasons, assembly routines are often needed to simplify the task of hardware-software interface tasks. Most common examples of assembly programs are programs for runtime startup to setup the environment for the execution of C-coded programs. A startup routine typically performs the following functions:

1. Load stack pointer (if using a stack)
2. Manipulation of hardware registers. Sometimes, a hardware register must be initialized within a certain time interval of power-up, and this can only be performed by an assembly routine.
3. Initialize global variables either by initializing the *automatic initialization block* in static RAM memory generated by the C-compiler for auto-initialized variables, or by using initialized values from a ROM.

The use of in-line assembly routines in C-programs simplifies the task of interfacing object code to the underlying processor hardware. A common example of in-line assembly is in enabling/disabling interrupts as shown by the Example below.

Example 5.9.15. Use of in-line assembly.

```
main()
{
        ...
        _asm("di\n");
        <critical code>
        _asm("ei\n");
}
```

□

As a matter of programming convenience, in-line assembly instructions need not use explicitly assigned processor registers. Most C-compilers allow the use of C-expressions as operands to assembly instructions. This allows us to use critical functions as assembly macros in C source programs as shown by the example below.

Example 5.9.16. C functions as assembly macros

```
#define sin(x) \
({double _value, _arg = (x) ; \
asm ("fsinx %1, %0" : "=f" (_value) : "f" (_arg)); \
_value; })
```

The assembly instruction `fsinx` uses C expression `x` as an operand. Type declaration `'f'` indicates that a floating point register must be used for this operand. A `'=f'` declaration indicates that output is a floating point register. The output operand `_value` must be a write-only *l-value*. □

5.10 Summary

While it is relatively straightforward to generate the actual program code from flow graphs, synthesis of software is complicated by the timing constraints. A model for the software and the runtime system is presented that consists of a set of program threads which are initiated by synchronization operations. Use of multiple program threads avoids need for a complete serialization of operations which may otherwise create $\mathcal{ND}$ cycles in the constraint graph.

Under timing constraints, linearization of a part of a flow graph may also not be possible. A heuristic linearization algorithm is suggested to perform operation linearization as a topological sort with a measure of urgency of scheduling operations based on timing constraint analysis on an unconstrained implementation.

The software generation is performed in steps, whereby first we create a linearized set of operations collected into program threads. The dependencies between program threads is built into the threads by means of additional enabling operations for dependent threads. Further additional overhead operations are added to facilitate concurrency between program threads either by means of subroutine calling, or as coroutines or as a program with case descriptions. Finally, the routines are compiled into machine code using compiler for the processor.

Chapter 6

System Partitioning

In this chapter we address the problem of partitioning of system functionality with the objective of achieving an implementation into separate components. The partitioning problem is of two types: *homogenous* or *heterogenous*. The objective of homogenous partitioning is to partition a system functionality into minimal number of parts such that all parts are implemented completely in hardware or software. Homogenous partitioning for hardware is typically done under size constraints on each of the parts, whereas for software implementations, the objective of partitioning is typically to increase resource utilization in order to achieve speedup in overall execution time.

We focus here on the heterogenous partitioning problem, where the objective is to partition the system model for implementation into hardware and software components. One approach to heterogenous partitioning would be to consider it as a generalization of the homogenous partitioning problem by treating the processor as a *generalized resource*, i.e., a resource that can implement multiple types of operations. However, there are inherent differences in the model of computation used for implementation of hardware and software models. As was discussed in Chapter 3 the software component implements model functionality as an *instruction-driven* computation with a statically allocated memory space. On the other hand, hardware components essentially operate as *data-driven* reactive components. Further, due differences in the primitive operations in hardware and software components, the two computations proceed at very different instantaneous rates of execution. Because of these differences in the models and rates of computation used

by hardware and software components, it is necessary to allow multiple executions of individual hardware and software models with respect to each other to achieve efficient hardware-software system implementation. Further, the difference in the rates of computations causes variations in the rates of communication between hardware and software components and thus entails a higher communication overhead than purely hardware or software partitions, due to necessary handshake and buffering mechanisms.

Clearly, the problem of partitioning into hardware and software is much more complex than partitioning for implementations into purely hardware or software. Often this partitioning is carried out at levels of abstraction that are higher than what can be modeled using conventional modeling schemes. In absence of requisite modeling capability, the system partitioning simply can not be carried out without human interaction. Thus, there exists a strong relationship between the models used for capturing system functionality and the abstraction level at which the partitioning is carried out. Fortunately, to a great extent partitioning at various levels of abstraction can be carried out independently so long as the objectives of partitioning are kept distinct. For example, a partition of functionality into electrical and mechanical components can be carried out without obviating a need for subsequent partitioning of the electronic components into multiple chips or boards. The partitioning procedure presented in this chapter attempts to perform a division of functionality *at the level of operations specified in the hardware description language* and is by no means a substitute for 'conceptual' partitioning usually carried out at higher levels of abstractions. Indeed, it attempts to supplement the conceptual design process by providing the system designer a means to handle the complexity associated with a detailed design description consisting of language-level operations.

Against this broad context, the problem of system partitioning refers specifically to a partition of the system model as described in Chapter 3. Recall that a system model consists of a set of flow graphs. A flow graph model refers to the hierarchy of flow graphs that implements a function or a process. This partitioning can be developed as a collective set of *transformations* on the flow graph model that achieve the correct separation of functionality as embodied by the flow graphs. Since the flow models are developed to allow estimation of performance and constraint analysis for hardware and software, this analysis can be used in conjunction with partitioning transformations to

ensure that partitioning objectives are met.

The partitioning problem for a flow graph refers to the assignment of operations in the graph to hardware or software. This assignment to hardware or software determines the delay of the operation. Further, the assignment of operations to a processor and to one or more application-specific hardware circuit involves additional delays due to *communication overheads*. Any good partitioning scheme must attempt to minimize this communication. Further, as operations in software are implemented on a single processor, increasing the number of operations in software increases the degree of utilization of the processor. Consequently, the overall system performance is determined by the effect of hardware-software partition on performance parameters that go beyond the area and delay attributes described in Chapter 3. Therefore, the key to achieving an effective partition is to develop an appropriate *cost model* that captures relevant performance parameters from the attributes of the flow graphs. This cost model is described next.

6.1 Partition Cost Model

The partition cost model is built upon the processor cost model Π and the software model using multiple program threads both described in Chapter 5. Figure 44 illustrates the software model shown in Figure 38 in Section 5.2 further and shows various components of the partition cost model and their relationship to the target architecture. The software component is characterized by following properties:

1. **Thread latency**, λ_i (seconds) indicates the upper bound on the execution delay of a program thread.

2. **Thread reaction rate**, ϱ_i (per second) is the invocation rate of the program thread. A bound on the reaction rate of the program thread is determined by the imposed *rate constraints* on the operations within the program thread.

3. **Size of software**, S^{Π} is the size of memory and data to implement the software.

The hardware component is characterized by its size. The hardware size, S_H is expressed in terms of number of cells needed to implement hardware using a specific library of gates. It provides a measure of the actual area of hardware implementation.

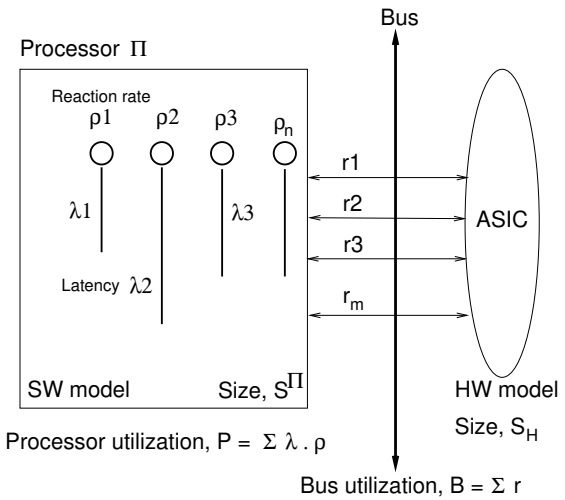


Figure 44: *Components of the partition cost model*

A hardware-software implementation is characterized by the following parameters.

1. **Processor utilization**, $\mathcal{P}$ indicates utilization of the processor. It is defined as

$$\mathcal{P} \doteq \sum_{i=1}^{n} \lambda_i \cdot \varrho_i \tag{6.69}$$

2. **Bus utilization**, $\mathcal{B}$ (per second) is a measure of the total amount of communication taking place between the hardware and software. For a set of m variables to be transferred between hardware and software,

$$\mathcal{B} \doteq \sum_{j=1}^{m} r_j \tag{6.70}$$

r_j is the inverse of the minimum time interval (in seconds) between two consecutive samples for variable j. This interval is determined by the rate constraint on the input/output operation associated with a variable.

A partition of the system model Φ is refers to a partition of the flow graphs in Φ into two groups such that one is implemented into hardware and the other into software. Since a flow graph may be partitioned into hardware-software implementations, this separation is modeled by transformations on the flow graph that generate two separate interacting

flow graphs. Since these transformations may, in general, require replication of vertices and edges in order to correctly model functionality, a partition of the flow graph is not exactly a partition of graphs in the mathematical sense, that is, there may be an overlap of vertices across partitions. This situation is not peculiar to hardware-software partitioning but a necessary side-effect of partitioning at a behavioral level of abstraction.

Given the partitioning cost model, the problem of partitioning a specification for implementation into hardware and software can then be stated as follows:

Problem P1: *Given a system model, Φ as set of flow graphs, and timing constraints between operations, create a partition $\varpi(\Phi) = \Phi_S \cup \Phi_H$ in to two sets of flow graph model, Φ_H and Φ_S such that a hardware implementation of Φ_H and a software implementation of Φ_S implements Φ and the following is true:*

1. *Timing constraints are satisfied for all flow graphs in Φ_H and Φ_S,*
2. *Processor utilization, $\mathcal{P} \leq 1$,*
3. *Bus utilization, $\mathcal{B} \leq \overline{\mathcal{B}}$. The bound on bus utilization is a function of bus bandwidth and memory latency.*
4. *A partition cost function,*

$$f(\varpi) = a_1 \cdot S_H(\Phi_H) - a_2 \cdot S^{\Pi}(\Phi_S) + b \cdot \mathcal{B} - c \cdot \mathcal{P} + d \cdot |m| \quad (6.71)$$

is minimized. Here $|m|$ defines the cumulative size of variables m that are transferred across the partition, and $a_1,\ a_2,\ b,\ c$ and d are positive constants.

Parameters a_1, a_2, b, c and d represent desired tradeoff among size of hardware, software implementation, processor and bus utilization and the communication overheads. Let us first consider the size metrics: S_H and S^{Π}. The size of hardware is computed from the size attribute of a operation.

$$S_H(\Phi_H) = \sum_{G_i \in \Phi_H} S_H(G_i) = \sum_{G_i \in \Phi_H} \sum_{v \in V(G_i)} S(v) \quad (6.72)$$

Since $S(v)$ is a local property of the vertex, it does not include hardware costs for control and scheduling logic circuits. Next, the software size is computed as

$$S^{\Pi}(\Phi_S) = \sum_{G_i \in \Phi_S} S^{\Pi}(G_i) = \sum_{G_i \in \Phi_S} [S_d^{\Pi}(G_i) + S_p^{\Pi}(G_i)]$$

We use the following (worst case) approximation for the program size as the number of *rvalues*, denoted by ω, associated with a vertex. That is,

$$S_p^{\Pi}(v) = O(\omega(v)) = O(|rvalue(v)|)$$

where $O()$ refers to the order-of approximation. This approximation is based on the observation that the number of instructions generated by a compiler would be related to the number of *rvalues*. Of course, global optimizations will reduce the number of instructions below this bound. However, use of this bound gives us a measure of the software size during partitioning phase. It should be pointed out that in some compilers, the number of instructions generated for a language-level operation may be more than *rvalues*, specially if certain *rvalues* lead to more than one assembly instructions. This may happen especially during transformation that change the underlying operator, for example, operator strength reduction. Still, the proportionality assumption for the program size to the number of *rvalues* is valid.

The data size $S_d^{\Pi}(v)$ is harder to calculate from operation-level attributes, since register allocation affects the total data storage globally. From Inq. 5.56 in Section 5.4 we have a bound on the data size as:

$$S_d^{\Pi}(G) \leq |M(G)| + |P(G)|$$

where $M(G)$ is the storage variables associated with G and $P(G)$ is the set of input and output ports of G. When partitioning a flow graph model into two models, to the first order we can assume that the data set $M(G)$ is replicated in both partitions in order to ensure correct functionality of the resulting partitioned graph models. The effect of a move of an operation on data storage is to change the set of input and output ports of G. Even though this change increases the software size, it does not help the goal of maximization of operations in the software. Instead, this change is accounted for as an

adverse effect on the quality of the partition by increasing its communication overhead, m. Therefore,

$$S_d^{\Pi}(v) = 0 \Rightarrow S^{\Pi}(\Phi_S) = \sum_{G_i \in \Phi_S} \sum_{v \in V(G_i)} O(w(v)) \tag{6.73}$$

The next three parameters in the cost function, $\mathcal{B}$, $\mathcal{P}$ and m are related to the communication cost of the partition and the number of operations in the program threads. All of these parameters can, therefore, be calculated from operation dependencies and their attributes in the flow graph. Let us assign a weight, $c(u, v)$ to the edge (u, v) as the size of data transfer from vertex u to vertex v. For our implementations, a control transfer from is simulated by means a data transfer on a port. The communication cost $|m|$ refers to the sum of edge weights over the edges that lie across the partition, ϖ.

The task of hardware-software partitioning requires evaluation and selection of operation vertices in the individual flow graphs. An exact solution to the constrained partitioning problem, that is, a solution that minimizes the partition cost function requires evaluation of a large number of operation groupings which is typically exponentially related to the number of operations in the system model. As a result, heuristics to find a 'good' solution are often used with the objective of finding an optimal value of the cost function. This optimality of the cost function is defined over grouping of operations achieved by some neighbourhood selection operations. Our heuristics to solving the partitioning problem starts with a constructive initial solution which is then improved by an iterative procedure by moving operations across the partition as explained by the following pseudo-code.

```
▷I    construct initial partition, ϖ ← ϖ0
      compute f(ϖ0)
      repeat {
▷II       select a group of operations to move
          create new partition, ϖ'
▷III      compute f(ϖ')
          if f(ϖ') < f(ϖ)
               accept move: ϖ ← ϖ'
      } until no more operations to move
```

Before describing the heuristics, let us first take a look at the nature of the cost function that is used to direct the heuristic search. The cost function consists of a set of *properties*, for example, $S_H, S^{\Pi}, \mathcal{P}, \mathcal{B}, m$ in our case. Since a large number of group of operations are possible candidates for iterative improvement, the computation of the cost function, $f(\varpi)$ significantly affects the overall time complexity of the procedure. A cost function consisting of properties that require evaluation of the entire graph model at each step of the iteration would add to the complexity of the search procedure. On the other hand, an ideal case would be a cost function that can be *incrementally* updated from previous value, that is, the change in the cost function can be computed quickly. Such a quick estimation also allows for variable-depth neighbourhood search methods also such as Kernighan-Lin [KL70] or Fidducia-Matheysis [FM82], where many hypothetical alternatives can be explored to select the 'best' group of operations to move. Let us first examine the type of properties that constitute a cost function.

6.2 Local versus Global Properties

We need to capture not only the effects of *sizes* of hardware and software parts but also the effect of *timing* behavior of these portions, as captured by processor and bus utilization, into the partition cost function. During partitioning iteration, a movement of an operation across the partition causes a change in its delay and therefore the latency of the flow graph.

In general, it is hard to capture the effect of a partition on timing performance during partitioning stage. Part of the problem lies in the fact that timing properties are usually *global* in nature, thus making it difficult to make incremental computations of the partition cost function which is essential in order to develop effective partition algorithms. For each local move, the timing properties must be calculated for the entire graph model, which adds to the computational complexity of the heuristic search process.

Traditionally, there exists a spectrum of techniques in partitioning, and use of timing properties in driving the partitioning search process. On one end of the spectrum are partitioning schemes for hardware circuits which are mostly focussed on optimizing area (and pinout) of resulting circuits and do not use timing properties. On the other end of

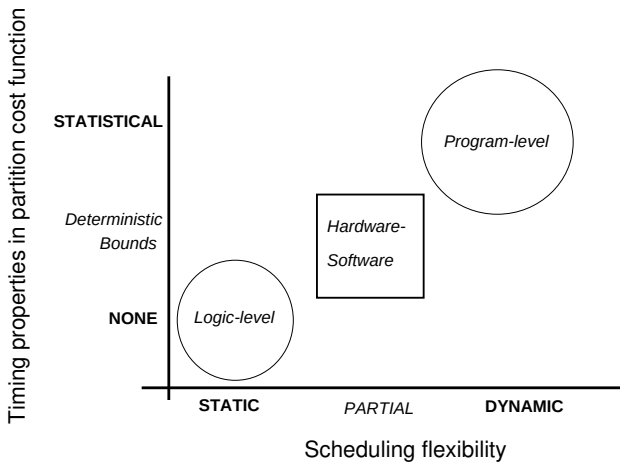


Figure 45: *Use of timing properties in partition cost function*

the spectrum are partitioning schemes in software world where the objective of partitioning is to create a set of programs to execute on multiple processors. These schemes do make extensive use of statistical timing properties in order to drive the partitioning algorithm [Sar89]. The distinction between these two extremes of hardware and software partitioning is drawn by the flexibility to schedule operations. Hardware partitioning attempts to divide circuits which implement scheduled operations, therefore, there is not much need to consider the effect of partition on overall latency which is to a great extent dependent upon the choice of schedule for the operations.[1] In contrast, the program-level partitioning problem addresses operations that are scheduled at run-time. Because of this ability to schedule operations at run time, the timing properties are more complex and dependent upon the operating environment and external input to the software. Thus, a statistical measure is often used to capture the timing properties of a partition.

As shown in Figure 45, we take an intermediate approach to partitioning for hardware/software systems, where we use deterministic bounds to compute timing properties that are incrementally computable in the partition cost function, that is, the new partition cost function can be computed in constant time. This is accomplished by using a software

[1]Partitioning for unscheduled flow graphs was considered in [GM90] that considers a cost function using latency and size properties. The update of the latency in the inner loop of search for the best group of operations was achieved by an approximation technique that allowed for incremental update of the latency, even though it is a global property.

model in terms of a set of program threads as shown in Figure 44 and a partition cost function, f, that is a linear combination of its variables. Thus, the characterization of software using λ, ϱ, $\mathcal{P}$ and $\mathcal{B}$ parameters makes it possible to calculate static bounds on software performance. Use of these bounds is helpful in selecting appropriate partition of system functionality between hardware and software. However, it also has the disadvantage of overestimating performance parameters. For example, the actual processor and bus utilization depends upon the distribution of data values and communication based on actual data values being transferred across the partition. Instead we determine the *feasibility* of a partition based on the worst case scenario and ensure that this worst case scenario is handled by the partition alternatives being evaluated by the partition cost function.

6.3 Partitioning Feasibility

A system partition into an application-specific and a re-programmable component is considered *feasible* when it implements the original specifications and it satisfies the performance constraints. We assume that the hardware and software compilation, done using standard tools, preserves the functionality. We, therefore, concentrate on constraints. There are two kinds of constraints that are used to determine feasibility of a partition:

1. Timing constraints as min/max delay and execution rate constraints, and

2. Performance constraints in terms of processor and bus utilization and ensuring that the a given runtime scheduler is able to meet constraints on software.

As mentioned earlier, when partitioning system model into hardware and software components the data rates may not be uniform across models. The discrepancy in data-rates is caused by the fact that the application-specific hardware and re-programmable components may be operated off different clocks and the system execution model supports multi-rate executions that makes it possible to produce data at a rate faster than it can be consumed by the software component when using a using a finite sized buffer. In presence of multi-rate data transfers, feasibility of hardware-software partition is determined by the fact that for all data transfers across a partition, the production and consumption

data rates are compatible with a finite and size-constrained interface buffer. That is, for any data transfer across partition, data consumption rate is at least as high as the data production rate. The production and consumption rates for a data transfer are defined by the reaction rate of the corresponding flow graphs.

6.3.1 Effect of runtime scheduler

The runtime scheduler refers to the *main* program in software that integrates calls to various program threads implemented as coroutines. As explained in Chapter 5, the runtime system used here mainly consists of a scheduler that invokes program threads at runtime. This scheduler can be of the following two types:

- **Non-preemptive runtime scheduler.** Here a program thread executes either to its completion or to the point when it detaches itself voluntarily (for example, to observe dependence on another program thread). Most common examples of non-preemptive schedulers are first-in-first-out (FIFO) or round-robin (RR) schedulers. FIFO schedulers select a program thread strictly on the basis of the time when it is enabled. A RR scheduler repetitively goes through a list of program threads arranged in a circular queue. A non-preemptive scheduler may also be *prioritized* or *non-prioritized*. Priority here refers to the selection of program threads from among a set of enabled threads. Both strict FIFO and RR maintain the order to data arrival and data consumption, no starvation is possible. A prioritized discipline may however lead to starvation. Alterations in scheduling discipline are then sought to ensure fairness, that is, the best prioritized discipline leads to least likelihood of a starvation.

- **Preemptive runtime scheduler**. These schedulers provide the ability to preempt a running program thread by another program thread. Preemption generally leads to improved response time to program threads at increased cost of implementation. This ability to preempt is tied to an assignment of priorities to program threads. The primary criterion in design of a preemptive scheduling scheme is in selecting appropriate priority assignment discipline that lead to most feasible schedules. Priority selection can be *static* where the priorities do not change at run time, for

example rate-monotonic priorities [LL73], or *dynamic*, for example deadline-driven priorities [LW82].

We have so far considered only non-preemptive runtime scheduling techniques. The reason for this is to keep the synthesized software simple. As was discussed in Chapter 5, the implementation of multiple program threads in a preemptive runtime environment leads to additional states for the program threads which adds to the overhead delay caused by the runtime scheduler. This is not to say that non-preemptive scheduling techniques are always sufficient for embedded systems. However, the very ability to do runtime scheduling of operations provides a substantial departure from static scheduling schemes used in hardware synthesis, and for the co-synthesis approach formulated here as an extension of high-level synthesis techniques, the choice of a non-preemptive runtime system provides a first step towards synthesizing embedded systems.

The basis for analysis of the runtime scheduler is provided by the intuition that its is sufficient to show the feasibility of the scheduler by considering the case when all threads are enabled at the same time. This observation has been used in analysis of several runtime scheduling algorithms and has been formalized by Mok [Mok83].

A necessary condition to ensure that the reaction rates of all program threads can be satisfied by the processor is given by the constraint that processor utilization is kept below unity, i.e.,

$$\mathcal{P} \leq 1 \tag{6.74}$$

However, this condition is not sufficient. Consider the following example.

Example 6.3.1. Program threads and processor utilization.

Consider a software consisting of two program threads, T_1 and T_2:

$$\lambda_1 = 10 \text{ cycles} \quad , \quad \varrho_1 = 0.01 = \frac{1}{100} \text{ per cycle}$$
$$\lambda_2 = 100 \text{ cycles} \quad , \quad \varrho_1 = 0.001 = \frac{1}{1000} \text{ per cycle}$$

The processor utilization, $\mathcal{P} = 10 \times 0.01 + 100 \times 0.001 = 0.2$ is well below unity. However, in the worst, using FIFO scheduling, thread T_1 is enabled after $10 + 100 = 110$ cycles. Thus supportable reaction rate for T_1 is 0.0091 or $\frac{1}{110}$ per cycle which is less than ϱ_1. □

For a program thread in a non-preemptive non-prioritized FIFO runtime scheduler, a sufficient condition to ensure satisfaction of its reaction rate, ϱ is given by the following condition:

$$\frac{1}{\varrho} \geq \sum_{\forall \text{ threads } k} \lambda_k \tag{6.75}$$

This inequality follows from the case when all the program threads are enabled simultaneously. In this case, a program thread is enabled again only after completing execution of all other program threads. Note that this condition is only sufficient. It is also necessary and sufficient for independent threads. In case of dependent program threads, only a subset of the total program threads are enabled for execution, that is, those threads that do not depend upon execution of the current thread. Therefore, the necessary condition will be weaker and can be estimated by summation over enabled program threads in Inq. 6.75.

From Inq. 6.75, a sufficient condition for software reaction rate satisfiability is to ensure that

$$\frac{1}{\varrho_{\max}} \geq \sum_{\forall \text{ threads } k} \lambda_k \tag{6.76}$$

where $\varrho_{\max} = \max_i \varrho_i$ defines the maximum reaction rate over all program threads. It is interesting to note that the same condition for worst case reaction rate also applies for RR schedulers, though the average case performance differs.

Remark 6.1 (Prioritized runtime scheduler) A prioritized FIFO scheduler consists of a *set* of FIFO buffers that are prioritized such that after completion of a thread of execution the scheduler goes through the buffers in the order of their priority. The program threads are assigned a specific priority ψ and are enqueued in the corresponding FIFO buffer. Thus, among two enabled program threads, the one with the higher priority is selected. The effect of this priority assignment is to increase the *average* reaction rates for the program threads with higher priority at the cost of decrease in the *average* reaction rate for the low priority threads. Recall that in a non-prioritized scheduler the supportable reaction rate is fixed for all threads as the inverse of the sum over all thread latencies. Unfortunately, the worst case scenario

gets considerably worse in case of a prioritized scheduler: it is possible that a low priority thread may never be scheduled due to starvation.

The average case performance improvement can be intuitively understood by the following analysis. Consider a software of n threads, $T_1,\ T_2,\ \ldots,\ T_n$ with reaction rates, $\varrho_1,\ \varrho_2,\ \ldots,\ \varrho_n$ respectively. Let $\psi(T_i)$ be the priority assignment of one of the levels from 1 to l, with l being the highest priority and 1 being the lowest priority. For each thread, let us define *background processor utilization* as

$$\eta(T_i) = \sum_{\forall\ T_k\ \ni\ \psi(T_k)<\psi(T_i)} \lambda_k \cdot \varrho_k \tag{6.77}$$

That is, $\eta(T_i)$ provides a measure of the processor utilization by the set of program threads with priority strictly lower than $\psi(T_i)$. Thus the *available* processor utilization for threads with priorities greater than equal to $\psi(T_i)$ is given by $1 - \eta(T_i)$. On an average, this can be seen as an extension in the latency of program threads. Let us define an *effective latency*, λ' as

$$\lambda'_i = \frac{\lambda_i}{1 - \eta(T_i)} \tag{6.78}$$

The average case feasibility condition is now defined as

$$\frac{1}{\varrho(T_i)} \geq \sum_{\text{thread}\ T_k\ \ni\ \psi(T_k)\geq\psi(T_i)} \lambda'_k \tag{6.79}$$

As an aside we note that a prioritized scheduler can be used in conjunction with a preemptive scheme. Here, the effect of priority assignment on supportable reaction rates can be estimated by considering a reduction in available processor utilization for a thread, caused by scheduling of higher priority threads. This reduction in processor utilization can be modeled as increase in the *effective* latency of the program threads, with the difference that the execution of a program thread is now spread over a non-contiguous time interval. For a review of the scheduling analysis using preemptive schemes please see [SR88].

To summarize, the notion of feasibility of a partition between hardware and software buids upon the feasibility of each of the two components and additional constraints on processor and bus utilization. To be specific, a flow graph, $G_i \in \Phi_H$ is considered feasible if it meets the timing constraints under the assignment of hardware operation delays and no runtime scheduler. A flow graph, $G_i \in \Phi_S$ is considered feasible if it meets the imposed timing constraints under the assignment of software operation delays and software storage operations and there exists a feasible linearization of operations in G_i under the timing constraints. Timing constraint satisfiability can be checked by the procedures *check_satisfiability* as described in Chapter 4. Similarly, linearization can be checked by the procedure outlined in Chapter 5. A partition of Φ in Φ_S and Φ_H is considered feasible if:

1. for all $G_i \in \Phi_H$, G_i is hardware feasible,
2. for all $G_i \in \Phi_S$, G_i is software feasible,
3. For all program threads T_i in software, Φ_S

$$\frac{1}{\max \varrho(T_i)} \geq \sum_k \lambda_k$$

   This condition also ensures that processor utilization is below unity.

4. Bus bandwidth, $\mathcal{B} \leq \overline{\mathcal{B}}$.

We now consider possible approaches to realizing a partition of the system model in to flow graphs for hardware and software.

6.4 Partitioning Based on Separation of Control and Execute Procedures

A partition of functionality can be obtained by considering the system model as consisting of interacting *control* and *execution* procedures. The execution procedures perform data manipulation where as the control procedures direct the flow of execution and data. There are two possibilities:

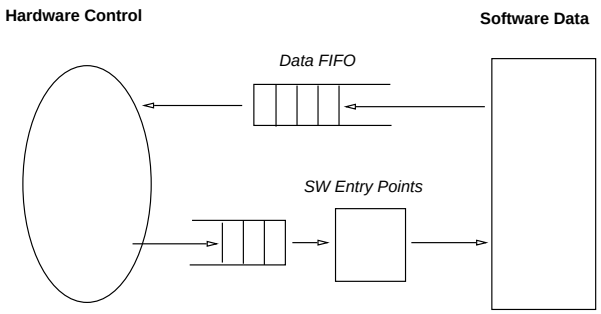


Figure 46: *Partitioning into Hardware Control and Software Execute Processes*

1. The hardware generates the address and data values for software execute process to start executing. In Figure 46 the software consists of a set of looping routines. The input data and loop counts are provided to the software addressing unit by the hardware.

2. The software control provides a mechanism to dynamically schedule hardware resources. This case is similar to microcoded machines where microcode uses different hardware resources to control flow of execution.

While promising, neither these approaches are considered further since the system model does not explicitly make a distinction between control and execute procedures. Thus separation of control and execute portion would require an extensive set of transformations to the flow graph model.

6.5 Partitioning Based on Division of $\mathcal{ND}$ Operations

Our procedure for partitioning a flow graph model is based on the iterative improvement procedure presented in Section 6.1. There are three main components to this procedure:

I. Creation of initial partition. The initial partition is constructed by creating program threads for each of the $\mathcal{ND}$ loop operations in the flow graph model. It is assumed that for all the external synchronization operations, a corresponding rate constraint is provided, which is used as a property of the environment with which the system interacts. Due to this property, rates of data transfer for all inputs to the software is

known. This rate of data transfer then defines the reaction rate of the corresponding destination program thread. On the other hand, the rate of data production from the software is determined by achievable reaction rate of the associated program thread. Feasibility of a partition is checked by applying the routine *check_feasibility*.

II. Selection of a group of operations to move across partition. Selection of operations requires a check for partitioning feasibility. Among the available vertices we pick operation vertices with known and bounded delay. With this vertex moved across the partition to software, its attributes are updated and the corresponding graph model is checked for constraint satisfiability by applying the procedure *check_satisfiability* in Chapter 4. If the move is a partitioning feasible more, the next vertex to be selected is one of the immediate successors of the vertex moved. This way, the group of vertices selected for a move constitutes path in the flow graph. This heuristic selection is made to reduce the communication cost.

III. Update of the cost function. After the initial computation of the cost function, changes to the cost function are computed incrementally. A vertex move from hardware to software, entails the following changes to the partition cost function:

$$\Delta f = a_1 \Delta S_H - a_2 \Delta S^{\Pi} + b \Delta \mathcal{B} - c \Delta \mathcal{P} + d \Delta |m| \qquad (6.80)$$

1. Hardware size S_H is reduced by the size attribute of the vertex according to Equation 6.72. So the reduction in cost function due to move of vertex, v into software is given by

$$\Delta f|_{S_H} = a_1 \cdot S(v)$$

2. Software size S^{Π} is increased by the *rvalue* attribute, ω of the vertex according to Equation 6.73. So the reduction in cost function is given by

$$\Delta f|_{S^{\Pi}} = a_2 \cdot \omega(v)$$

Note that the effect of an operation move into software is to increase the size of the software which decreases the cost function.

3. The change in communication cost, Δm, is computed by examining the neighbours of vertex, v. This is also used to compute the change in bus utilization, $\Delta \mathcal{B}$.

4. Processor utilization, $\mathcal{P}$ is computed by considering two cases. One where the reaction rate of the destination thread is unaffected. In this case the reduction in cost function due to vertex move, v to thread k is given by

$$\Delta f|_{\mathcal{P}} = c \cdot \varrho_k \delta(v)$$

where $\delta(v)$ refers to the software delay of operation v. In case, the operation move changes the reaction rate of a thread to ϱ'_k, the effect on cost function reduction is given by

$$\Delta f|_{\mathcal{P}} = c[\varrho'_k \delta(v) + (\varrho'_k - \varrho_k)\lambda_k]$$

The algorithm to perform graph-based partitioning is described by the following pseudo-code. Starting with a system model, it examines flow graph models that in Φ corresponding to each process model for possible partition. Procedure *graph_partition* returns a *tagged* graph indicating its partition into two graphs. Vertices in a tagged graph are labeled according to their partition membership. This graph is then subject to partition transformations described later in Section 6.6 such that the resulting graphs correctly implement the specified functionality.

Input: System model, $\Phi = \{G_i\}$, Processor model, Π, Bus bandwidth $\overline{\mathcal{B}}$, Runtime overhead, $\overline{\gamma}$
Output: Partitioned system graph model, $\Phi = \Phi_S \cup \Phi_H$

```
partition(Φ) {
    Φ_S = Φ_H = ∅;
    for each process graph in G_i ∈ Φ {
        graph_partition(G_i);                          /* returns a tagged graph, G_i */
        (G_H, G_S) = partition_transformation(G_i);    /* create separate graphs */
        Φ_S = Φ_S ∪ {G_S};
        Φ_H = Φ_H ∪ {G_H};
    }
}
```

```
graph_partition(G) {
    V_H = V;
    V_S = ∅;
▷I  for v ∈ V(G) {                          /* create initial partition */
        if v is an NCD link operation
            V_S = V_S + {v};
    }
    create software threads (V_S);          /* create initial |V_S| routines */
    compute reaction rates, ϱ for each thread;  /* based on rate constraints */
    if not check_feasibility(V_H, V_S)      /* no feasible solution exists */
        exit ;
    f_min = f(V_H, V_S);                    /* initialize cost function */
    repeat {
        for vertex v ∈ V_H and v is not NCD {   /* pick a det. delay operation from hw */
▷II         f_min = move(v);                /* select operations to move to sw */
} until no further reduction in f_min;
    return(V_H, V_S);
    }
}
move(v) {                                   /* consider v for move from V_H to V_S */
    if check_feasibility(V_H − {v}, V_S + {v})
▷III    if Δf > 0
            V_H = V_H − {v};                /* move this operation to sw */
            V_S = V_S + {v};
            f_min = f_min − Δf;
            update software threads;
            update thread reaction rate of the destination thread;
            for u ∈ succ(v) and u ∈ V_H     /* identify successor for move */
                move(u);
    return f_min;
}
check_feasibility(Φ_H, Φ_S) {               /* check partition feasibility */
    for all G_i ∈ Φ_H                       /* check timing constraints for hardware */
        if not check_satisfiability(G_i);
            return not feasible;
    for all G_i ∈ Φ_S                       /* check timing constraint for software */
        if not check_satisfiability(G_i);
            return not feasible;
    for all T_i ∈ Φ_S
        if (1 / max ϱ(T_i) < Σ_k λ_k)       /* runtime scheduler */
            return not feasible;
    if B > B̄                                /* check bus utilization */
        return not feasible;
    return feasible;
}
```

The markers $\triangleright$ indicate the main steps of the algorithms as described earlier in the beginning of this section. The algorithm uses a greedy approach to selection of vertices for move into Φ_S. There is no backtracking since a vertex moved into Φ_S stays in that set throughout rest of the algorithm. For each vertex move, the change in the partition cost function, Δf is computed in constant time. The constraint satisfiability algorithm *check_satisfiability* computes the all pair longest paths in the constraint graph which using Floyd's algorithm runs in $O(|V|^3)$ time. Since this satisfiability check is done for each possible vertex move, therefore, the complexity of the algorithm is $O(|V|^4)$.

6.6 Partition Related Transformations

From a partition of vertices in the input flow graph, a set of behavior-preserving graph transformations are applied in order to generate resulting interacting flow graphs. These transformations implement the abstract cut of the graph model. The following transformations are applied:

Replication - refers to replication of a vertex. This usually done to preserve the behavior of the conditional paths to a vertex.

Insertion of IBC vertices - An edge that crosses a partition represents a communication between hardware and software. Depending upon the reaction rate of the producing and consumer graph models, this communication can be either blocking, nonblocking or buffered. Implementation of these communications is discussed in Section 7.2.2. Appropriate interblock communication (IBC) vertices are added to support the communication and synchronization between operations across partitions.

In order to preserve the original sequencing dependencies for correct behavior, when the called model is partitioned into two sets of operations it is essential to duplicate the link vertex corresponding to the two model calls as shown in Figure 47. The link vertex, C, is duplicated into two link vertices, $C1$ and $C2$ each of which calls the respective graph model. Note that dependency edges $(a,c),(c,b),(d,c),(c,e)$ are modified accordingly in order to preserve the original sequencing dependencies.

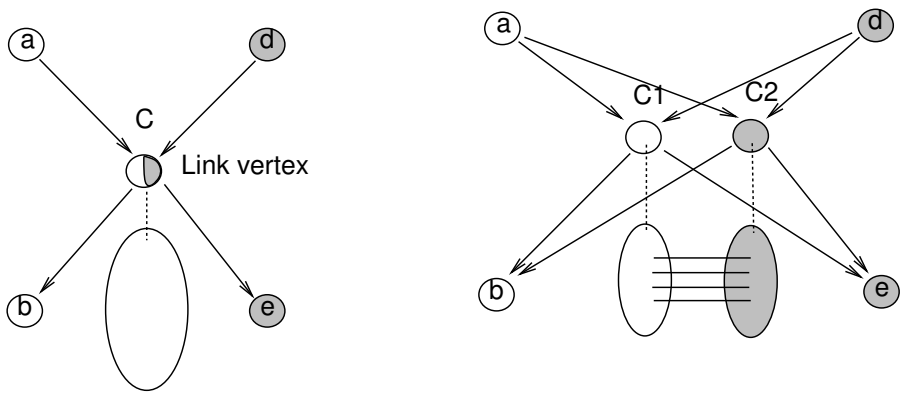


Figure 47: *Partition of link vertices*

When the partitioned vertices that lie on conditional paths, it is important to maintain the semantics of the conditional vertex by ensuring that the conditional logic associated with the path to the operation is properly replicated in both partitions. In case of link vertices related to loops, these vertices also contains logical conditions to either terminate the call. In this case also the link vertex replication with additional control edges is necessary in order to preserve the original behavior. Partitioning of a condition vertex is achieved by duplicating first the condition vertex (which represents the operation that computes the conditional clause) and corresponding join vertex in the (well-formed) flow graph. The original and replicated vertices are assigned to different partitions.

6.7 Summary

Partitioning of a system model into hardware and software is a difficult problem due the local and global properties of the costs involved in partitioning. Our approach to partitioning is characterized by development of a suitable partition cost model that makes it possible to incorporate both size and performance parameters into the partitioning objective function, while at the same time allowing efficient incremental computation of the cost function. This capability achieved by means of using deterministic bounds on performance parameters such as processor and bus utilization using worst case analysis. A disadvantage of this approach is the possible overestimation of the utilization parameters. Based on our analysis of the runtime scheduler, we develop constraints on partitions that

ensure feasibility of the partition to meet imposed constraints. This feasibility makes use of the notion of constraint satisfiability as developed in Chapter 4 and proceeds by performing graph analysis on individual hardware or software constraint graph models. Our partitioning procedure is one of building an initial partition based on loop $\mathcal{ND}$ operations, and then developing an iterative improvement procedure that uses partitioning feasibility in selecting operations to be moved across the partition while maintaining satisfiability of the imposed timing constraints.

Chapter 7

System Implementation

Hardware-software cosynthesis is not a single task but consists of a series of tasks. These tasks are related to modeling of functionality and constraints, analysis of constraints, model transformations to ensure constraint satisfiability, partitioning of the model and partitioning-related transformations, synthesis of hardware and software components, simulation of the final system design. These subtasks have been implemented in a general *framework*, called VULCAN, that allows user interaction at each step of the cosynthesis process and guides the system designer to the goal of realizing a mixed system design. This chapter discusses the implementation of the Vulcan system and its relationship to other tools to accomplish synthesis and simulation of hardware-software systems.

Further, the target architecture presented in Section 1.7.1 of Chapter 1 is simple and leaves open many different possible ways of implementing the hardware-software interface and communication mechanisms. We present the architectural choices made by Vulcan and possible extensions and alternatives. We conclude this chapter by a discussion of our approach to the cosimulation of hardware-software systems.

7.1 Vulcan System Implementation

Vulcan is written in the C programming language and consists of approximately 60,000 lines of code. Through its integration with the Olympus Synthesis System [MKMT90] and DLX processor compilation and simulation tools [HP90], it provides a complete path

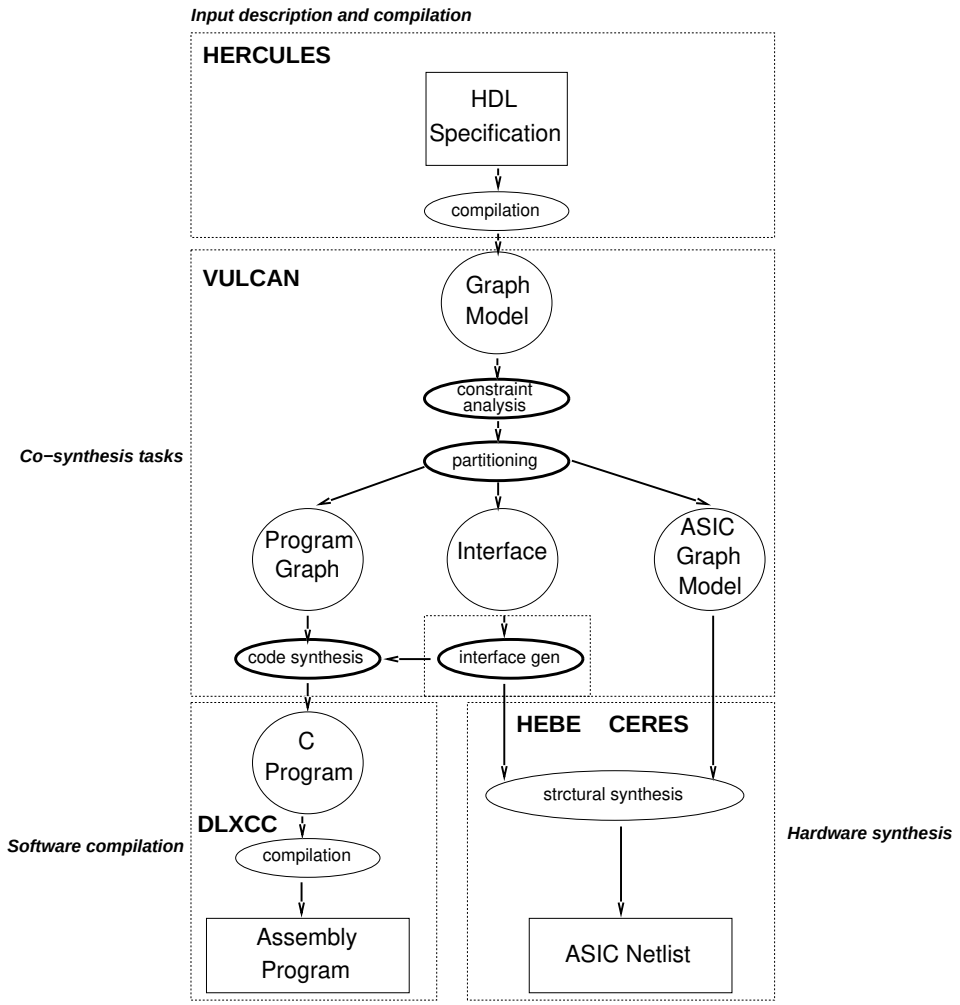


Figure 48: *Co-synthesis flow.*

for synthesis of hardware and software form *HardwareC* descriptions. A block diagram of the co-synthesis flow was shown in Figure 11 in Chapter 1 and is reproduced in Figure 48 for convenience.

The input to Vulcan consists of two components: a description of system *functionality* and a set of *design constraints*. The design constraints consists of timing constraints and constraints on parameters used during the co-synthesis process. Timing constraints are specified along with the system functionality in *HardwareC* by means of the *attribute* mechanism. These attributes make use of statement *tags* that identify the operation subject to constraints.

Example 7.1.1. HardwareC description is annotated by the following attribute commands to specify minimum and maximum execution rate constraints, identify loop index variables, and specify clock names and cycle times.

.attribute "constraint <minrate—maxrate> [<num>] of <tag> = <num> cps"	Rate constraints
.attribute "loop-index <str> [<num>]"	Index variable
.attribute "clock <str> [<num>]"	Clock signal
.constraint mintime from <tag> to <tag> = <num> cycles	Min delay
.constraint maxtime from <tag> to <tag> = <num> cycles	Max delay

□

This input is first compiled into a sequencing graph model (SIF) using the program HERCULES [KM90b] by applying a series of compiler-like transformations. The sequencing graph model is then translated into the bilogic flow graph model by performing the following operations:

1. Identify signal wait operations: these operations are specified as loop operations with an empty loop body, e.g., "`while(reset);`" which defines just a (busy-wait) implementation of the corresponding wait operation. The signal wait operations are unimplemented by defined as an atomic `wait` operations.

2. Merge SIF graphs on conditional hierarchies.

3. Identify storage variables for each graph body.

4. Classify loop operations as pre- or post-indexed.

Some guidelines must be observed when specifying HardwareC inputs for cosynthesis purposes. The arithmetic operations must be bound to resources in order to prevent Hercules from generating combinational logic operations to implement the respective operations. This can be done by mapping and binding arithmetic operations to specific library functions or to 'dummy' operators which are translated into respective operators without associated function calls. This limitation has to do with the fact that even though the semantics of SIF is general enough to support arithmetic operations on multi-bit variables, the hardware synthesis of these operations must be carried out by explicit function/procedure calls to specific library modules.

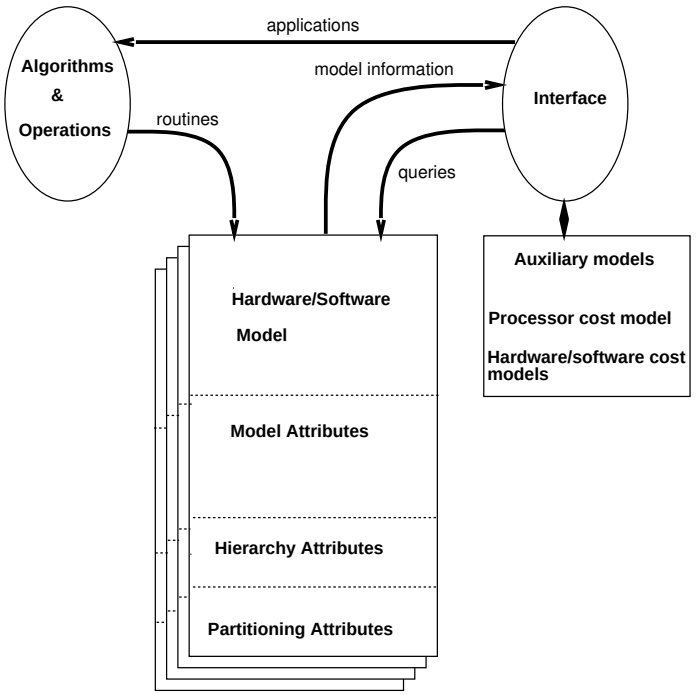


Figure 49: *Data Organization in* VULCAN

7.1.1 Data organization in Vulcan

The organization of data in Vulcan is shown in Figure 49. Vulcan maintains a list of hierarchically connected graph models. The flow graph models may be implemented or unimplemented. In addition, Vulcan maintains a list of processor cost models and cost models for implementation of software and hardware. The format of the processor cost model is described in Appendix C. The cost model for hardware and software store the results of actual hardware and software synthesis and are updated by the mapping results from hardware synthesis, and by parsing the dissembler output respectively. The algorithms for analysis and transformations are applied on these graph models. At this time, all transformations of the graph model are user driven. The model manipulation routines automatically update the list of models after transformations and update the cost models and attributes with the result of analysis.

7.1.2 Command organization in Vulcan

Vulcan provides an interactive menu-driven interface that is modeled after the Unix™ shell and was originally developed by Frederic Mailhot [MKMT90]. This interface provides the typical shell commands related to directory and file management, input/output redirection, aliasing and history commands. Three levels of command complexity is supported and command abbreviation is provided for advanced users. In addition, the interface provides a 'freeze' command that saves the state of Vulcan system and data into a dump file that can be restored later in case there is a need to back to some previous step of the co-synthesis process. This user interface is supported across all the tools in the Olympus Synthesis System, thus making it convenient for the user to move among the tools in the same session.

Figure 50 shows the organization of Vulcan subsystems and their relationship to hardware and software synthesis. Vulcan consists of following sub-systems. For each of these sub-systems a command menu is presented that lists the commands relevant to the subsystem. A history stack maintains the subsystems being used by the user. This allows for excursions into different subsystems as needed (for example, model manipulations related to constraint analysis may use synthesis results instead of using estimation routines).

1. **Model maintenance and manipulations** (command `Enter_model`) supports manipulations and constraint satisfiability analysis on the flow graph model. The maintenance functions include reading and writing of graph models, and cost model for the processors, identification of data and control dependencies and types of loop operations.

2. **Model partitioner** (command `Enter_partitioner`) Partitioning is accomplished by first 'tagging' a flow graph model, followed by creation of individual flow graph models through partitioning transformations. Assignment of graphs is made on the basis of partitioning feasibility analysis that checks for constraint satisfiability of hardware and software implementations of the individual flow graphs, and feasibility of the runtime scheduler to support the hardware and software portions based on processor and bus utilization.

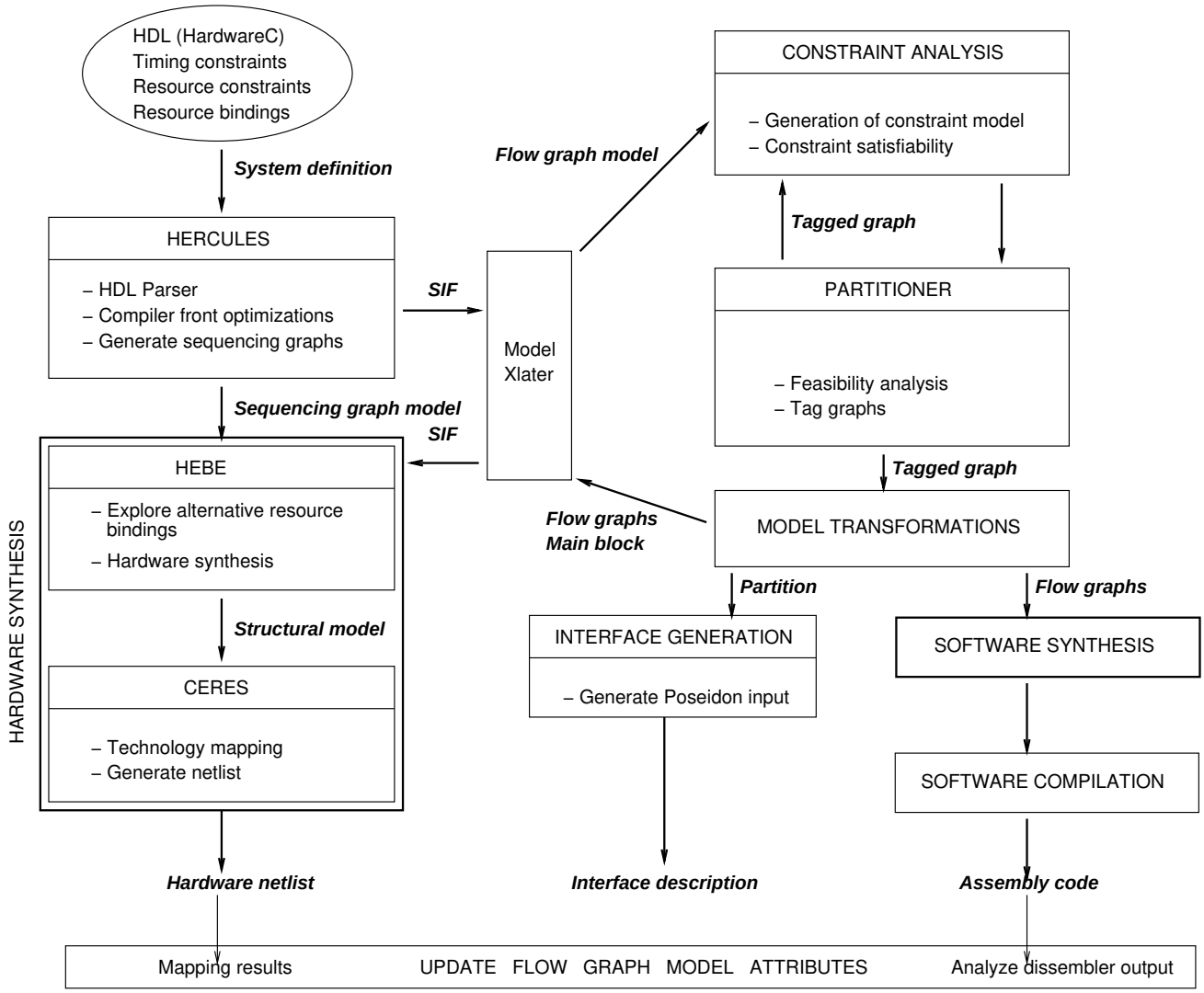


Figure 50: *Vulcan subsystems and the Olympus Synthesis System*

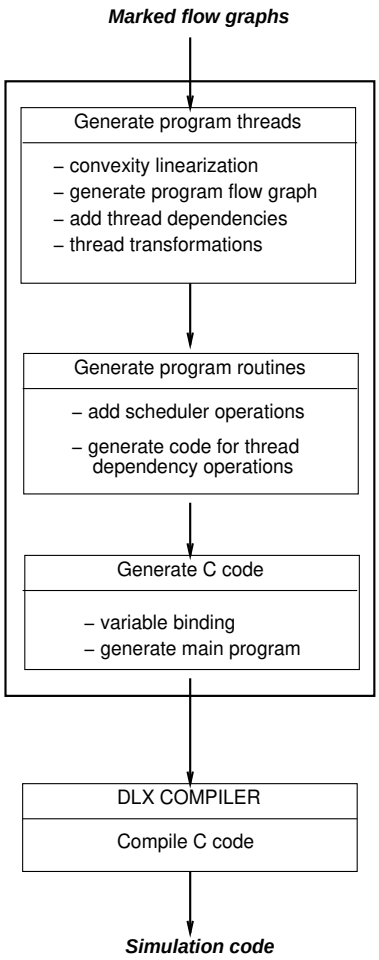


Figure 51: *Flow of software synthesis in Vulcan*

3. **Hardware synthesis** (command `Enter_hardware`) is performed by passing the corresponding sequencing graph model to programs HEBE and CERES in the Olympus synthesis system.

4. **Software synthesis** (command `Enter_software`) consists of tasks of program thread identification, serializations, generation of routines. Figure 51 shows the command flow in generation of the software component. The software synthesis also performs generation of scheduling routines (enqueue, dequeue) and hardware-software interface routines (transfer_to) for the runtime system.

5. **Interface synthesis and model simulations** (command `Enter_interface`) Interface description is currently entered manually from specified models and interface protocols. Simulation of the mixed hardware-software system is performed by the program Poseidon described in Section 7.3.

The supported commands in each of the subsystem are listed in Table 7.

7.2 Implementation of Target Architecture in Vulcan

As mentioned earlier, there are several issues that must be resolved in order to bring the target architecture described in Section 1.7.1 closer to a realization. Among the important issues in iits implementation are use of communication and synchronization mechanisms and the architecture of the interface between hardware and software components. We assume that the communication across hardware-software is carried out over a communication bus. We further assume that only the processor is the bus master, thus obviating a need for implementation of bus arbitration logic in the dedicated hardware.

7.2.1 System synchronization

System synchronization refers to mechanism for achieving synchronization between concurrently operation hardware and software components. Due to pseudo-concurrency in the software component, that is, concurrency simulated by means of operation interleaving, a data transfer from hardware to software must be explicitly synchronized. Using a *polling* strategy, the software component can be designed to perform *pre-meditated transfers* from the hardware components based on its data requirements. This requires static scheduling of the hardware component so that the software is able to receive the data when it needs it. In cases where the software functionality is communication limited, that is, the processor is busy-waiting for an input-output operation most of the time, such a scheme would be sufficient. Further, in the absence of any $\mathcal{ND}$ operations, the software component in this scheme can be simplified to a single program thread and a single data channel since all data transfers are serialized. However, this would not support

Command	*Description*
Model maintenance and manipulations	
buildrp	Build resource utilization pool
check_satisfiability	Constraint satisfiability analysis
dataflow	Extract data-flow dependencies
flatten	Flatten a flow graph
hierarchy	Display hierarchy
list, rename, delete, print	Model maintenance
readsif, readcpu	Read sequencing model, processor cost model
Model partitioner	
augment	Augment a model with partitioning info
check_feasible	Partitioning feasibility analysis
cut	Cut a tagged flow graph into two graphs
cuthier	Show cut hierarchy
readattr, writeattr	Read/write partitioning attributes
setp	Set partitioning parameters
kl	KL based partitioning heuristic to tag graphs
untag	Untag a partitioned graph
Hardware synthesis	
eval_logic	Evaluate hardware cost using HEBE and CERES
mkblock	Make an interconnection block for partitioned models
synthesize	Pass model to HEBE and CERES for synthesis into hardware
readcost	Read synthesis cost model from HEBE
Software synthesis	
linearize	Linearization heuristic under timing constraints
blinearize	Perform a breadth-based linearization heuristic
dlinearize	Perform a depth-based linearization heuristic
estim_spill	Determine the spill set
estim_delay	Estimate delay for software implementation of a flow graph
mkconvex	Convexity serialization for a flow graph
mkthreads	Generation threads
mkmain	Write runtime scheduler
packboolean	Pack storage of boolean variables
printcode	C-code translation from flow graph
read/write_asm	Read/write an assembly file
read/write_dis_asm	Read/write a dissembler output
Interface	
write_poseidon	Write interface description to POSEIDON

Table 7: *Vulcan (Rev 0) subsystems and commands.*

any branching nor reordering of data arrivals since dynamic scheduling of operations in hardware would not be supported.

In order to accommodate different rates of execution of the hardware and the software components, and due to $\mathcal{ND}$ operations, we look for a *dynamic* scheduling of different threads of execution. Such a scheduling is done based on the availability of data. This scheduling is by means of a **control FIFO** structure which attempts to enforce the policy that the data items are consumed in the order in which they are produced. The hardware-software interface consists of data queues on each channel and a FIFO that holds the identifiers for the enabled program threads in the order in which their input data arrives. The control FIFO depth is sized with the number of threads of execution, since a program thread is stalled pending availability of the requested data. Thus the maximum number of places in the control FIFO buffer would be the maximum number of threads in the system. Example 7.2.2 below shows an example of the interface between hardware and software.

Example 7.2.2. Hardware-Software Interface

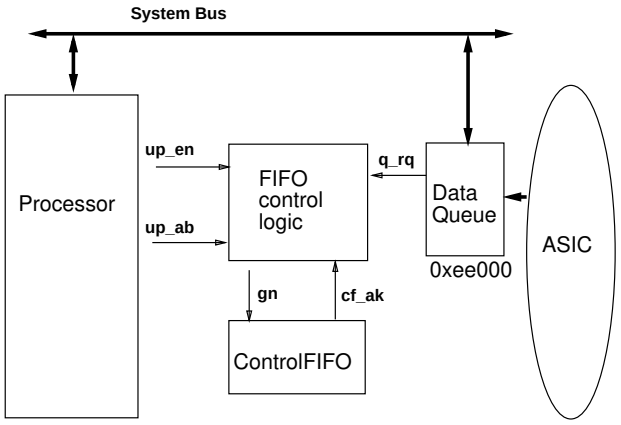


Figure 52: *Control FIFO schematic*

Figure 52 shows schematic connection of the FIFO control signals for **a single data queue**. In this example, the data queue is **memory mapped** at address 0xee000 while the data queue request signal is identified by bit 0 of address 0xee004 and enable from the microprocessor (up_en) is generated from bit 0 of address 0xee008.

The control logic needed for generation of the enqueue is described by a simple state transition diagram shown in Figure 53. The control FIFO is ready to enqueue

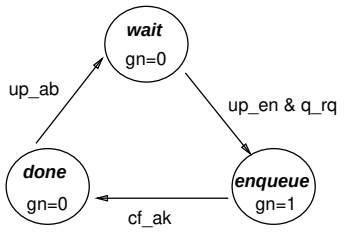


Figure 53: *FIFO control state transition diagram*

(indicated by `gn = 1`) a process id if the corresponding data request (`q_rq`) is high and the process has enabled the thread for execution (`up_en`). Signal `up_ab` indicates completion of a control FIFO read operation by the processor.

In case of multiple fanin queues, the `enqueue_rq` is generated by OR-ing the requests of all inputs to the queues. In case of multiple-fanout queues, the signal `dequeue_rq` is generated also by OR-ing all dequeue requests from the queue. □

The control FIFO and associated control logic can be implemented either in hardware as a part of the ASIC component or in software. In the case that the control FIFO is implemented in software, the FIFO control logic is no longer needed since the control flow is already in software. In this case, the `q_rq` lines from the data queues are connected to the processor unvectored interrupt lines, where the respective interrupt service routines are used to enqueue the thread identifier tags into the control FIFO. During the enqueue operations, the interrupts are disabled in order to preserve integrity of the software control flow. The protocol governing the enqueue and dequeue operations to the control FIFO are described using guarded commands in a interface description file that is input to the system co-simulator described in Section 7.3. Example 7.2.3 below shows a specification for the control FIFO based on two threads of execution.

Example 7.2.3. Specification of the control FIFO based on two threads of execution.

```
queue [2] controlFIFO [1];
queue [16] line_queue [1], circle_queue [1];

when ((line_queue.dequeue_rq+ & !line_queue.empty) & !controlFIFO.full) do
controlFIFO enqueue #1;
when ((circle_queue.dequeue_rq+ & !circle_dequeue.empty) & !controlFIFO.full)
do controlFIFO enqueue #2;
when (controlFIFO.dequeue_rq+ & !controlFIFO.empty) do controlFIFO dequeue
dlx.0xff000[1:0];

dlx.0xff000[2:2] = !controlFIFO.empty;
```

In this example, two data queues with 16 bits of width and 1 bit of depth, `line_queue` and `circle_queue`, and one queue with 2 bits of width and 1 bit of depth `controlFIFO` are declared. The guarded commands specify the conditions on which the number 1 or the number 2 are enqueued – here, a '+' after a signal name means a positive edge and a '-' after the signal means a negative edge. The first condition states that when a request for a dequeue on the queue `line_queue` comes and the queue is not empty and the queue `controlFIFO` is not full, then enqueue the value 1 in the `controlFIFO`. The last command just specifies a direct connection between signal `not controlFIFO.empty` and bit 2 of signal `dlx.0xff000`. □

7.2.2 Communication protocols

The hardware-software interface protocol is classified as one of *blocking*, *non-blocking* or *buffered*. A blocking communication protocol is expressed as a sequence of simpler operations on ports and additional control signal to implement the necessary handshake. For example, to implement a blocking read operation on a channel '`c`' additional control signals '`c_rq`' and '`c_ak`' would be needed as shown in the Example below.

Example 7.2.4. A blocking read operation.

```
bread(c)          =>      [
                          write c_rq = 1;
                          wait(c_ak);
                          < read(c);
                          write c_rq = 0; >
                          ]
```

□

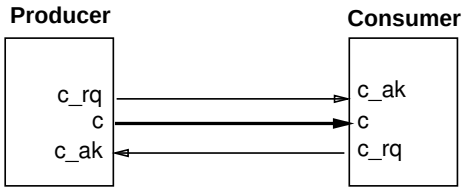
While it is easy to connect two blocking or two non-blocking read-write operations, connection of two disjoint read/write operations on a channel requires handling of special cases. For example, consider a connection between blocking read and non-blocking write operations below.

Example 7.2.5. Blocking/Non-blocking channel connections.

Blocking read and non-blocking write

```
    Blocking read                    Non-blocking write

[                                    [
```




```
write c_rq = 1;                          write c_rq = 1;
wait(c_ak);                              < write c = value; write c_rq = 0; >
< read(c); write c_rq = 0; >             ]
]
```

Blocking write and non-blocking read

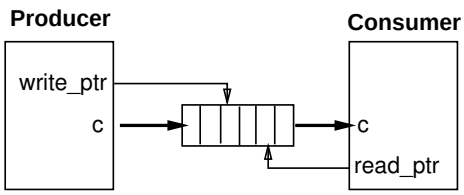
```
      Non-blocking read                  Blocking write

[                                        [
write c_rq = 1;                          write c_rq = 1;
< read(c); write c_rq = 0; >             wait(c_ak);
]                                        < write c = value; write c_rq = 0; >
                                         ]
```

A non-blocking/non-blocking read/write connection results in one cycle read and write operations. However, a blocking/non-blocking connection requires two clock cycles for the non-blocking operation. □

A buffered communication is facilitated by a finite-depth interface buffer with corresponding read and write pointers. The communication protocol consists of I/O operation as well as manipulation of the read, write pointers as shown by the example below.

Example 7.2.6. Buffered communication protocol.




```
[                                        [
read (buff[read_ptr]);                   write buff[write_ptr] = value;
read_ptr++ modulo N;                     write_ptr++ modulo N;
]                                        ]
```

Under normal operation, `read_ptr` $\neq$ `write_ptr`. Violation of this condition indicates either a buffer is full or empty depending on whether the increment of `write_ptr` causes violation or the increment of `read_ptr` causes the violation. □

7.2.3 Hardware-software interface architecture

The choice of the hardware-software interface protocol depends on the corresponding data transfer requirements imposed on the system model. In the case of known data-rates where (non-blocking) synchronous data transfers are possible, the interface contains an interface buffer memory for data transfer. A different policy-of-use for the interface buffer is adopted when transferring data or control information across the hardware-software partition. Therefore, the interface buffer consists of two parts: a data-transfer buffer and a control-transfer buffer (Figure 54). The data-transfer buffer uses an associative memory with *statically determined* tags, while the control-transfer buffer uses a FIFO policy-of-use in order to dynamically schedule multiple threads of execution in the software. Associated with each data-transfer we assign a unique tag which consists of two parts, software thread id and the specific data-transfer id. Since all the threads and all input/output operations are known, the tags are determined statically. The tag of a thread can be, for example, its entry point in the memory in case of a ROM code. In addition, the data-buffer contains a request flag (RQ bit) associated with each tag to facilitate the demand scheduling of various threads in software. Figure 55 explains the *modus operandi* of data transfer across a hardware-software partition. In the software, a thread of execution is in the compute state as long as it has all the available data as shown in Figure 55(a). In case of a dependency on another program thread or a graph model in hardware, the corresponding RQ bit is raised and the thread is detached as shown Figure 55(c). The processor then selects a new thread of execution from the control FIFO as shown in Figure 55(b). In case of data arrival to the interface buffer, if the corresponding RQ bit is on, its tag is put into the control FIFO as shown in Figure 55(c).

Note that the interface architecture described here shows only a *mechanistic* view of the hardware-software synchronization concepts presented before. Its implementation may be made simpler and yet achieve the same effect. For example, the functionality of the associative memory buffer can be translated into a software thread while using a simpler memory structure.

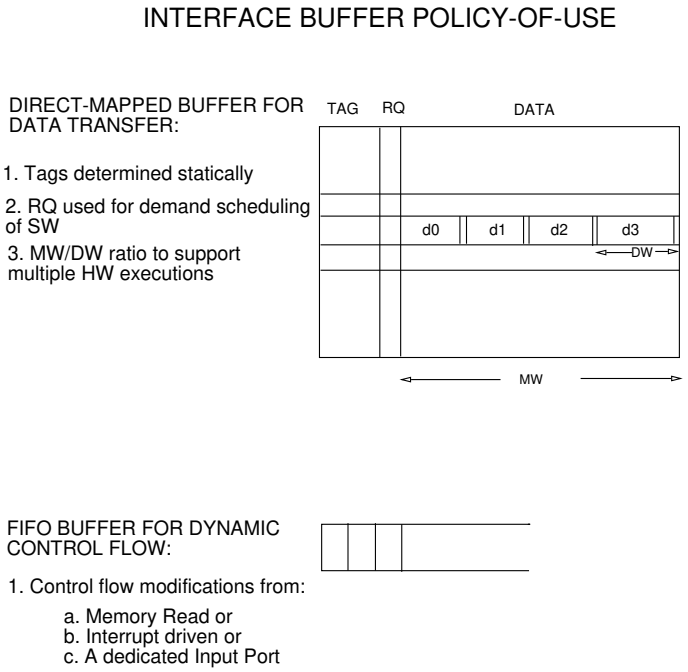


Figure 54: *Hardware and Software Interface Architecture*

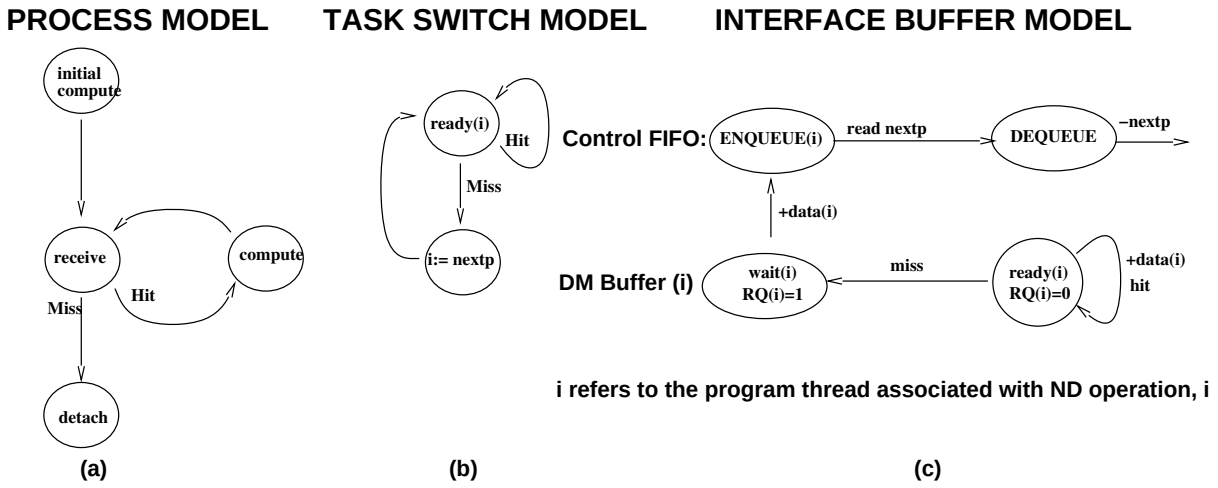


Figure 55: *Hardware and Software Interface Model*

7.3 Co-simulation Environment.

In this section, we briefly review major simulation concepts and techniques followed by a presentation of our approach to simulation of hardware-software systems.

Most simulators fall into one of the two categories: continuous or discrete event simulators. Continuous simulations occur frequently in control and systems engineering. In the context of underlying synchronous digital components, we are interested in discrete event simulations. In discrete event simulation, a simulation model of the system is exercised based on events on the inputs. Most discrete event simulators maintain a time-ordered queue of events. The queue may be centralized in a synchronous discrete event simulation or it may be distributed based on an asynchronous discrete event simulation. Examples of event-driven simulators using a global time scale are most simulators used for VHDL language [Sha86]. A frequent alternative to dynamic scheduling of events in discrete event simulation is *compiled code* simulation [WHPZ87]. In some circles, it is also known as a statically-scheduled or an oblivious simulator. In a compiled code simulation, there is no dynamic selection of events, as events are scheduled statically by a preprocessing step before the simulation begins. This avoids the overheads associated with management of event queue and event dispatch in event driven simulations at the potential cost of increased number of component evaluations. This can be done, for example, by treating components in a VHDL description as subroutines and their interconnection as variables. The resulting code can then be simulated by merely following the execution of the compiled code without the need for detailed event queues. This approach, also lacks detailed simulation information which may be needed to capture the so-called 'transient events'.

Simulation of a system consisting of interacting hardware and software components faces a practical problem in concurrent simulation due to a large disparity in the time scales over which *relevant* hardware and software actions are defined. An event driven simulation will seem to obviate this problem since it only simulates a network or component only when some events are generated, irrespective of the actual time scales. However, in practice large number of events are generated at the smallest interval of time granularity, hence a discrete event simulation is excessively slowed down due to its

need to evaluate all the events. A common approach to handling complexity in concurrent simulations is to perform a *process-oriented* simulation as opposed to event-oriented simulations. A process-oriented simulation can be thought of as a level of abstraction above event-orient simulations [Fis91] where the input specification in terms of concurrent processes is eventually *compiled* into an event-oriented simulation. This approach, however, does not make actual simulations any faster.

As a result of the above-mentioned practical problems in simulation of large systems, the design of a fast simulator applicable to co-simulation of hardware and software systems is an active area of research [BHLMar] [OH93].

We use program POSEIDON [GCM92b] that provides a practical environment for co-simulation of multiple functional models. Figure 56 shows the organization of the simulator. The input to Poseidon consists of specification of a collection of functional models and their associated simulators. Also, associated with each model, is a clock signal that is specified as an in-phase multiple of a common clock signal. Thus a hardware-software system is assumed to be centrally clocked.

The models specified in Poseidon can be implemented either in hardware or software. The software component is compiled into the assembly code of the target processor. Poseidon currently supports simulation of the assembly code for the DLX microprocessor. The hardware component can be simulated either before or after structural synthesis phase by using their respective simulators.

Poseidon carries out the hardware-software simulation by concurrently executing respective simulators for different input models. This is achieved by invoking each of the individual simulators at every cycle of the basic system clock, of which all other clock are a multiple. It maintains a queue of events which stores all simulation events on specified signals sorted by their activation times. After simulating an event, the resulting events are enqueued in the simulation queue. For each simulation cycle, all of the different simulators are invoked.

An input specification to Poseidon consists of following parts:

1. **Model declarations**: These consist of declarations of the concurrently executing simulation models. Models can be either software or hardware models. Each model has an associated clock signal and clock cycle-time used for its simulation. It is

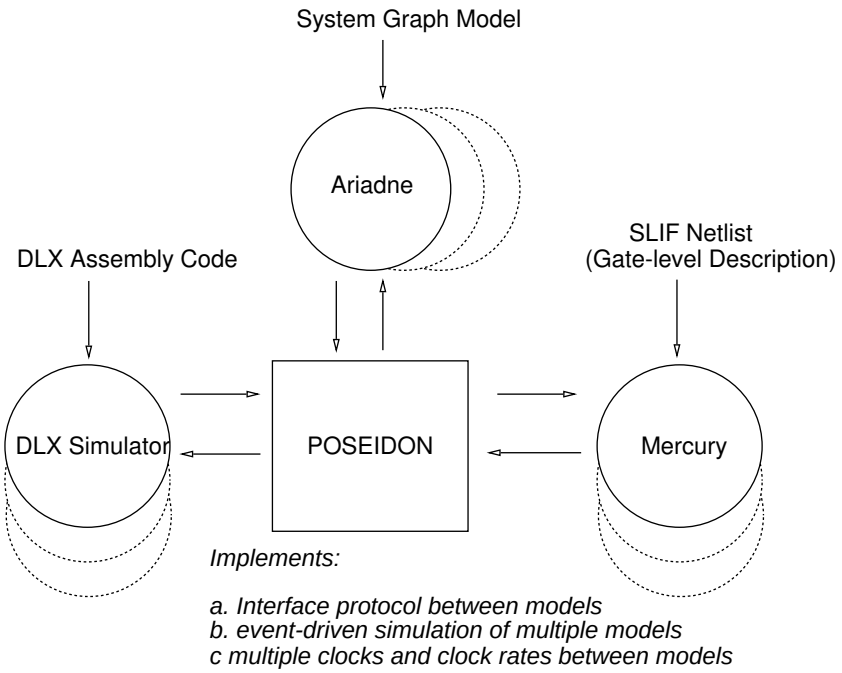


Figure 56: *Event-driven Simulation of a Mixed System Implementation*

assumed that the clock cycle-times are a rational multiple of a basic system clock. Further it is assumed that different models supply (latch) data at the interface using flip-flops at the interface edge-triggered by their respective clock signals.

2. **Model interconnections**: The interface between different system components is specified by *connections* among models. A connection between two models may be either a direct connection through a wire, or a port connection through a register or a queue. Queues can have multiple fanins and fanouts. Signal assignments indicate direct connections between respective models. For connections such as queues that require existence of additional control signals for synchronization, it is possible to group signals having identical synchronization requirements together for a given set of synchronization signals.

3. **Communication protocols**: Interface protocol for data-transfer is specified via *guarded* commands [Dij75]. A guarded command is a command which is executed only when some precondition is true. Each precondition is specified as a logic

equation of signal values and transitions. There are four commands recognized by the connection types. *Enqueue* and *dequeue* are used for queues port connections and *load* and *store* are used for register port connections.

4. **System outputs**: Outputs to be observed during simulation runs may be indicated by direct connections to the internal signals in the system model.

For illustration purposes, we consider a simple example of two models, `Producer` and `Consumer` that are connected by means of a finitely sized queue as shown in the Figure 57 in the following example.

Example 7.3.7. A producer-consumer system.

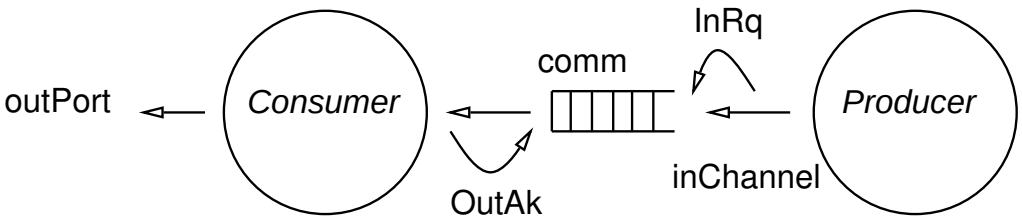


Figure 57: *Producer consumer system.*

Let us consider the case when the the producer model is implemented in software and the consumer model in hardware. The follow lists the interface description for this implementation.

```
# Models
model IO io 1.0 /local/ioDir IO;
model P dlx 1.0 /local/ProducerDir Producer;
model C mercury 3.0 /local/ConsumerDir Consumer;

# Connections
queue [4] comm[3];
C.RESET = IO.RESET;
C.r[0:0] = IO.r[0:0];

# Communication protocol
P.0xff004[0:0] = !comm.full;
C.b_rq = !comm.empty;
when (P.0xff000_wr+ & ! comm.full) do comm[0:3] enqueue P.0xff000[0:3];
when (C.b_ak+ & ! comm.empty) do comm[0:3] dequeue C.b[0:3];

# Outputs
IO.inChannel[0:3] = P.0xff000[0:3];
IO.outPort[0:3] = C.c[0:3];
IO.InRq = P.0xff000_wr;
IO.OutAk = C.b_ak;
```

The three first lines of the specification declare the models to be simulated. Model `io` models the external system inputs and outputs. The following parameter specifies

the clock period of the clock signal associated with the respective model. A value of 3.0 for the consumer model indicates that consumer is implemented in an ASIC technology that uses a clock signal that is three times slower than the clock used by the reprogrammable component, which is usually a custom designed component. The system input/outputs are sampled here at the same rate as the consumer. The last two parameters specify the directory location where the model description can be found and the model name. The `queue` statement declares a queue named, `comm`, which is 4 bits wide and 3 words deep. We use `rq` and `ak` signals to implement a blocking communication protocol as indicated by the guarded commands. A '+' suffix indicates rising edge transition of the corresponding signal. A '-' suffix indicates falling edge transition. Symbols '&' and '!' indicate the Boolean *and* and *not* operations.

The remaining commands are related to the interconnection of the interface. The meaning of an assignment is as follows: *bind the r-value coming from the output of some event,queue or register to the input of the l-value of the assignment*. The first assignment, `P.RESET = C.RESET = IO.RESET;`, for example, binds the signal `RESET` coming from IO to the signal `RESET` going to P and C. The last assignments specify the signals which will be seen at the end of the simulation.

Figure 58 shows *Poseidon* simulation results for the case when the software producer model is slower than the consumer model implemented in hardware. As shown in Figure 57, `inChannel` refers to the output of the producer model, while `outPort` refers to the output of the consumer model. As expected, consumer being the faster process is always ready for the new data by asserting the signal `OutAk`.

Figure 59 shows the simulation results for the case when the consumer model is slower than the producer model. In this case a three-deep queue is rapidly filled slowing down the enqueuing of data. □

As mentioned, Poseidon provides cycle-by-cycle simulation of concurrent models. This approach to hardware-software co-simulation in Poseidon has the advantage of simulating and verifying accurate relationships in time ordering of operations across models. It is also necessary in the context of our target system architecture that uses the same bus for interface to memory and ASIC hardware. Because of this commonality, explicit cycle-by-cycle simulations of all transactions over the common bus are required in order to be able to simulate the entire system. However, it has the disvantage of long simulation times since it simulates the mixed system at every cycle-step.

A more efficient model of simulation would be to use a distributed clock system in which the individual clocks perform local synchronizations. The primary advantage in such an *event-driven* co-simulation the individual simulation time-scales may not be

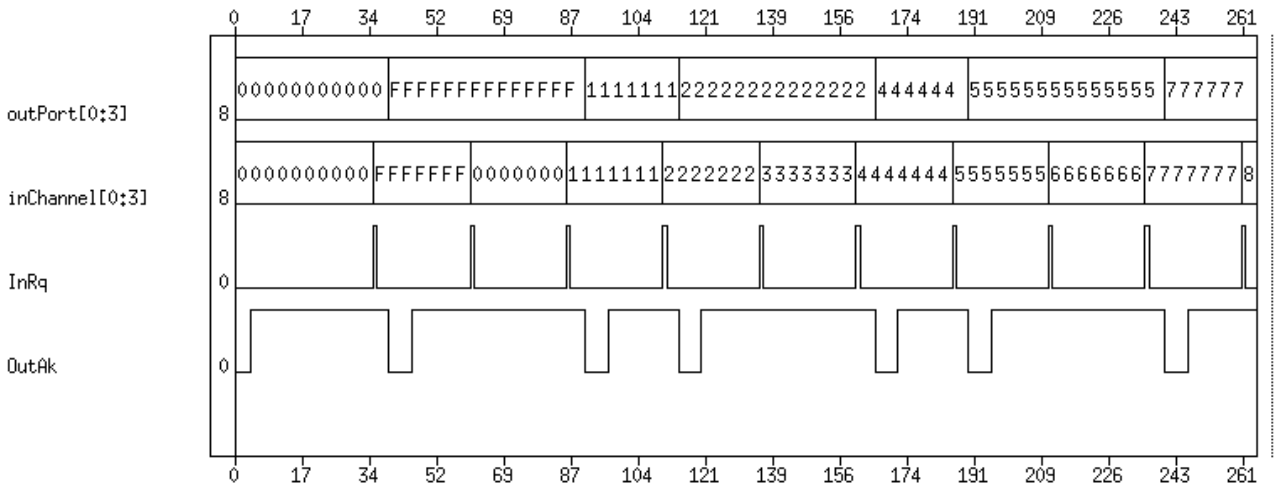


Figure 58: *Example simulation: software producer, hardware consumer*

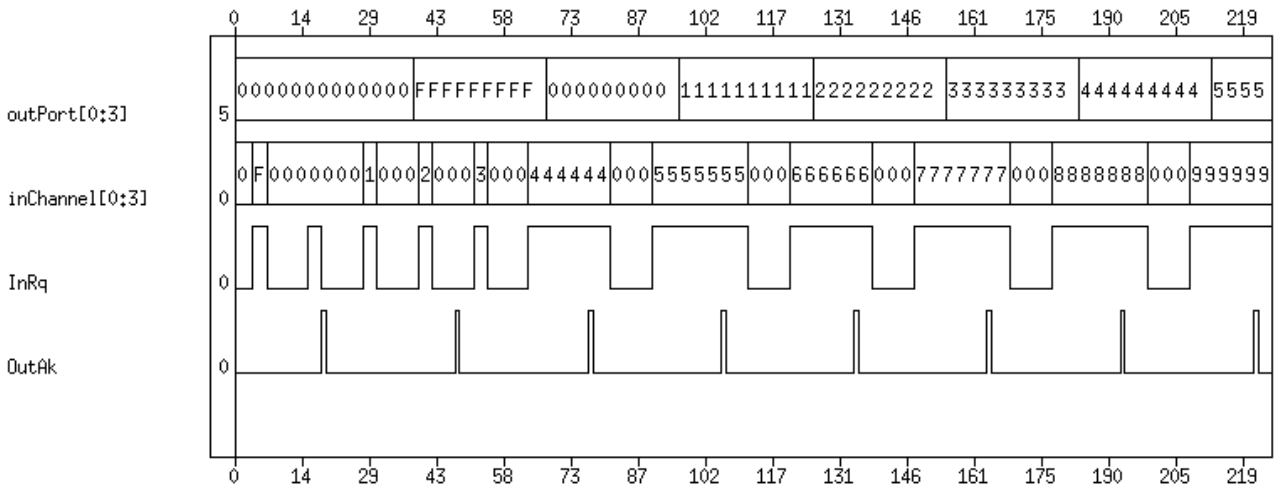


Figure 59: *Example simulation: software consumer, hardware producer*

synchronized. Recent work has shown the limited possibility of distributed clock event-driven simulations for hardware-software systems [tHM93]. In order to make such a simulation possible, it will be necessary to separate and hide the processor-level events from events related to hardware-software interactions. This is important since there may be numerous processor-level events that have very little or no impact on the interaction of the processor with the dedicated hardware.

We conclude by noting that the choice of the target architecture also determines the type of simulator needed for co-simulation A target architecture such as used in this work, exposes the hardware components to events on the common system bus.

7.4 Summary

In this chapter, we have presented the Vulcan framework that allows explorations into the system co-synthesis by evaluating hardware and software alternatives and their respective constraint satisfiability as developed in previous chapter. Due to the choice of a simple target architecture, many possible system realizations are possible. We have presented our choice of system implementation and organization of the interface and hardware-software synchronization mechanisms. Co-simulation of mixed systems remains to be a hard problem due to disparity in the time scales over which relevant events for hardware and software defined. Alternative means of achieving co-simulation are discussed.

Chapter 8

Examples and Results

This chapter presents results of system co-synthesis for benchmark examples. We present following two case studies in hardware-software co-design and compare hardware-software implementations against purely hardware implementations:

Graphics controller design. The purpose of the graphics controller is to provide a dedicated controller for generating actual pixel coordinates from parameters for different geometries. The input to the controller is a specification of the geometry and its parameters, such as end points of a line. The current design handles drawing of lines and circles. However, it is a modular design, where additional drawing capabilities can be added. The controller is intended for use in a system where the graphics controller accepts input geometries at the rate of 200 thousand per second, and outputs at about 2 million sample per second to a drawing buffer that is connected to a (CRT) device controller. Typically the path from the drawing buffer to the CRT controller runs at a substantially higher rate of about 40 million samples per second.

Network controller design. This controller implements the functionality of a carrier-sense, collision-detection protocol for handling multiple accesses over a shared communication medium. The controller works under specified timing constraints. However, a deterministic resolution of the timing constraints is difficult due to non-deterministic operations involved in handling multiple, variable-length data

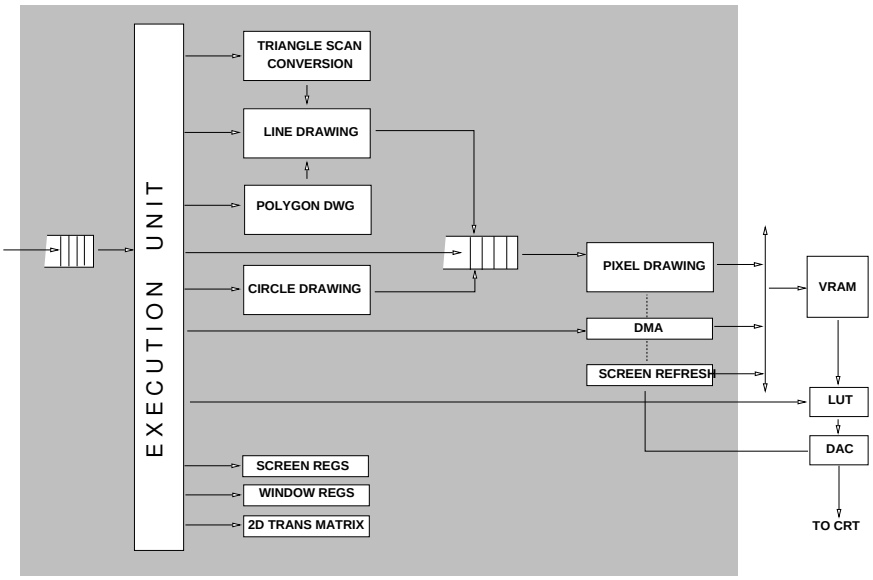


Figure 60: *Graphics controller block diagram*

packets.

8.1 Graphics Controller

Figure 60 shows the architecture of the graphics controller. The controller outputs pixel coordinates for lines and circles given the end coordinates (and radius in case of circle). The input to the controller is a queue of coordinates that are picked by the controller as soon as the previous drawing is finished. The rate at which these coordinates are picked up defines the input data rate. At the output of the controller is a video random access memory (RAM) buffer that provides for a high bandwidth path to the CRT controller.

8.1.1 Implementation

A mixed implementation of the controller design consists of line and circle drawing routines in the software component while the ASIC hardware performs initial coordinate generation and coordinates the transfer of data to the video RAM. The software component consists of two threads of execution corresponding to the line and circle drawing routines.

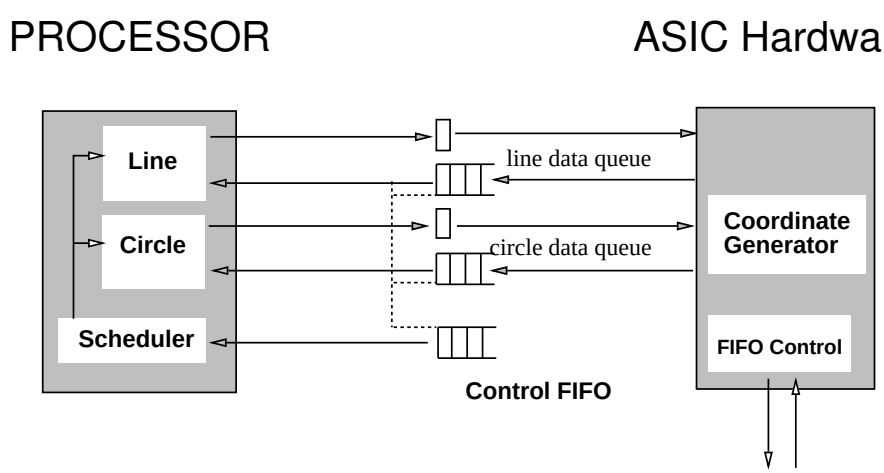


Figure 61: *Graphics controller implementation*

Both program threads generate coordinates that are used by the dedicated hardware. The data-driven dynamic scheduling of the program threads is achieved by a 3-deep control FIFO. The circle and line drawing program threads are identified by id numbers 1 and 2 respectively. The program threads are implemented using the coroutine scheme described in Section 5.7.1.

Figure 62 shows the main program in case of a hardware control FIFO implementation. Like the line and circle drawing routines, this program is compiled using existing C-compiler. The transfer routines are coded manually. Appendix D lists the transfer routines for hardware and software implementations of the control FIFO buffer.

Table 8 compares the performance of different program implementations using control FIFO either in hardware or in software component. The hardware implementation of a control FIFO with fanin 3, when synthesized into hardware and mapped to LSI 10K library of gates, costs 228 gates. An equivalent software implementation adds 388 bytes to the overall program size of the software component. Note that the cost of hardware control FIFO increases as the number of data queues increases. On the other hand, software implementation of control FIFO using interrupt routines (as described in Section 5.6.1) to perform the control FIFO enqueue operations offers lower implementation cost for a 50% increase in the thread latencies.

In case of a software implementation of control FIFO, the enqueue and dequeue

```
#include "transfer_to.h"

int lastPC[MAXCOROUTINES] = {scheduler, circle, line,main};
int current=MAIN;

int  *controlFIFO_out = (int *) 0xaa0000;
int  *controlFIFO = (int *) 0xab0000;
int *controlFIFO_outak = (int *) 0xac0000;

#include "line.c"
#include "circle.c"

void   main(){
          resume (SCHEDULER);
};

int nextCoroutine;

void scheduler() {
          resume (LINE);
          resume (CIRCLE);
          while (!RESET) {
               do {
                    nextCoroutine = *controlFIFO;
                } while ((nextCoroutine & 0x4) != 0x4);
                resume (nextCoroutine & 0x3);
           }
}
```

Figure 62: *Graphics controller software component using hardware control FIFO*

operations are written in C programming language, which are then compiled for DLX assembly. Figure 63 shows the main program in case of a software control FIFO. The overhead due to enqueue and dequeue operations is reduced further by manually optimizing the assembly code as indicated by the entry 'Opt. Software CFIFO'. This one time optimization of enqueue and dequeue routines, which does not affect the C-code description of the program threads, leads to a reduction in the program size and program thread overhead to 29.4% thereby improving the rate at which the data is output. Note that data input and output rates have been expressed in terms of number of cycles it takes to input or output a coordinate. Due to a purely data-dependent behavior of program threads, the actual data input and output rates would vary with respect to value of the input data. In this example simulation, the input rate has been expressed for a simultaneous drawing of a line and 5 pixel radius with width of 1 pixel each and the results are accurate to one pixel. An input rate of 81 cycles/coordinate corresponds to approximately 0.25 million samples/sec for a processor running at 20 MHz. Similarly, a peak circle output rate of 30 cycles/coordinate corresponds to a rate of 0.67 million samples/sec.

```
#include "transfer_to.h"

int *int1_ak = (int *) 0xb00000;
int *int2_ak = (int *) 0xc00000;

int controlFIFO[16];                                    /* Definition of queues */
int queuein=0, queueout=0, empty=1, full=0;

enqueue(id)
        int id;
{
       queuein = (queuein + 1) & 0xf;
       controlFIFO[queuein] = id;

       empty = 0;
       full = (queuein == queueout);
}

dequeue()
{
       queout = (queout + 1) & 0xf;

       full = 0;
       empty = (queuein == queueout);
       return controlFIFO[queueout];
}

int lastPC[MAXCOROUTINES] = {scheduler, circle, line,main};
int current=MAIN;

#include "line.c"
#include "circle.c"

void   main(){
           resume (SCHEDULER);
};

int nextCoroutine;

void scheduler() {
           resume (LINE);
           resume (CIRCLE);
           while (1) {
                while (empty);
                 transfer_to (dequeue());
             }
}
```

Figure 63: *Graphics controller software component using software control FIFO*

Scheme	*Program size* (bytes)	*Synchronization overhead delay* (% cycles)	*Input data rate*$^{-1}$ (cycles per coordinate)	*Output data rate*$^{-1}$ (cycles/coordinate)			
				line		circle	
				ave.	peak	ave.	peak
Hardware CFIFO	5972	0	81	535.2	330	76.4	30
Software CFIFO	6588	50	95	749.5	502	106.8	31
Opt. Software CFIFO	6360	29.4	95	651	407	94	31

Table 8: *A comparison of control FIFO implementation schemes*

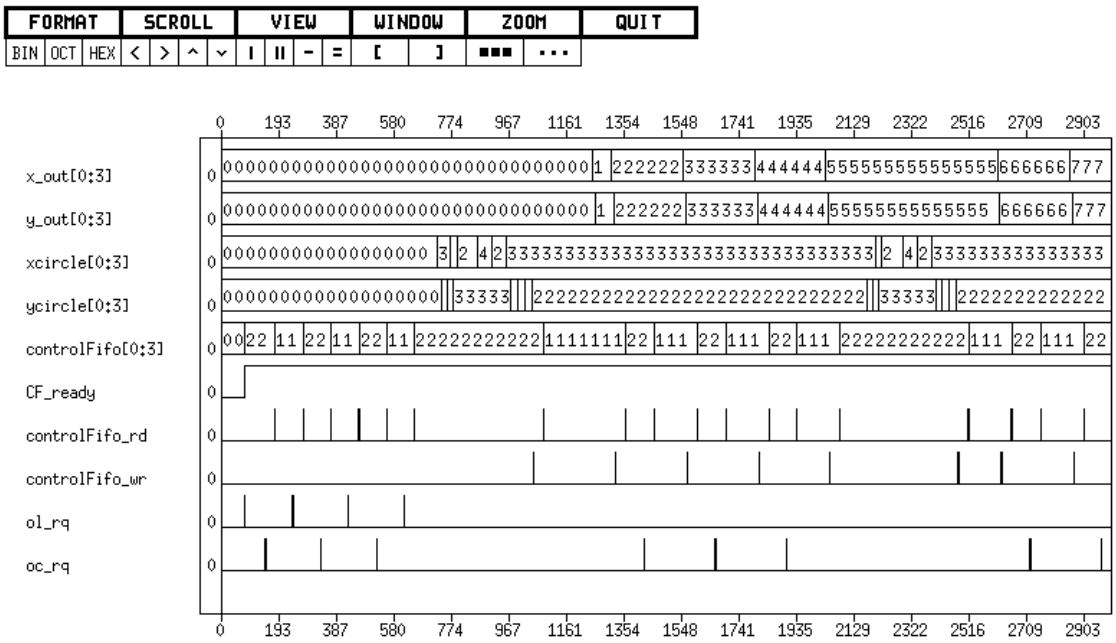


Figure 64: *Graphics controller simulation*

Implementation	*Size*	*Performance*
Complete hardware implementation	10,642 gates	14.70
Mixed implementation	228 gates, 5972 bytes	0.25

Table 9: *Graphics controller implementations.*

Figure 64 shows a simulation of the mixed implementation. `x_out` and `y_out` are the coordinates generated by the line thread routine. `xcircle` and `ycircle` are the coordinates generated by the circle thread routine. Note that these latter coordinates are generated in burst mode, since the circle thread routine explores symmetries to generate the coordinates. The values at the top of the `controlFIFO` are also shown in the figure. `CF_ready` signals that the `controlFIFO` is never empty after initialization. We show also the synchronization between the data queues, the lines and circle threads and the scheduler. `controlFIFO_rd` shows when the scheduler polls the `controlFIFO` to obtain the next thread id. `controlFIFO_wr` shows the transfer of control-flow from the line and circle threads. Finally, `ol_rq` (`oc_rq`) shows when the data fifo for the line (circle) enqueues the corresponding thread ids to signal that new coordinates are already available.

Table 9 presents a comparison of hardware and mixed implementation of the controller. Performance here is related to the input rate expressed in million samples/sec. Performance of a pure software implementation of the controller depends strongly upon the choice of the runtime system. A conventional subroutine based scheduler would add substantial overheads due to storage management operations. On the other hand, a software control FIFO implementation can be treated as a form of pure software implementation (using interrupts) which gives an input rate of 0.21 million samples/sec for a software size of 6360 bytes.

8.2 Network Controller

The network controller manages the processes of transmitting and receiving data frames over a network under CSMA/CD protocol, commonly used in Ethernet networks. CSMA/CD refers to Carrier Sense Multiple Access with Collision Detection protocol used to facilitate communication among many stations over a shared medium (or channel). It is defined by IEEE 802.3 standard. Briefly, CS means that any station wishing to transmit 'listens' first and defers its transmission until the channel is clear. MA implies simultaneous accesses by multiple stations is allowed without the use of any central arbitration. CD refers to collision detection protocol used to detect simultaneous transmission by two or more stations.

The purpose of this controller is to off-load the host CPU from managing communication activities. The controller contains two independent 16 byte wide receive and transmit FIFO buffers. The controller provides a small repertoire of eight instructions that let the host CPU program the machine for specific operations (transmit some data from memory, for example). The controller provides following functions:

- Data Framing and De-Framing
- Network/Link Operation
- Address sensing
- Error Detection
- Data Encoding
- Memory Access

8.2.1 Host CPU-controller interface

Both the CPU and the controller share a bus which can be controlled either by CPU or by the controller. The exclusivity of bus-master is ensured by handshake signals used between the two. The shared bus consists of all Address and Data lines.

In additions to CPU and controller, the bus is also connected to system memory. The controller contains a PC which contains the address from where its next instruction fetch occurs.

8.2.2 Controller operation

A typical controller operation consists of the following steps:

1. host cpu invokes the controller by write and a memory mapped address,
2. the controller responds by making a request for bus control,
3. once acknowledged the controller initiates memory read operation to receive command operations,
4. once initialized the controller relinquishes control of the bus to host cpu.

In the event of a **collision**, the controller manages the 'jam' period, random wait and retry process by re-initializing the DMA pointers without CPU intervention. In case of any **errors** in the received data, the controller re-initializes the DMA pointers and reclaims any data buffers containing the bad frame. All the transmitted and received data is manchester encoded/decoded.

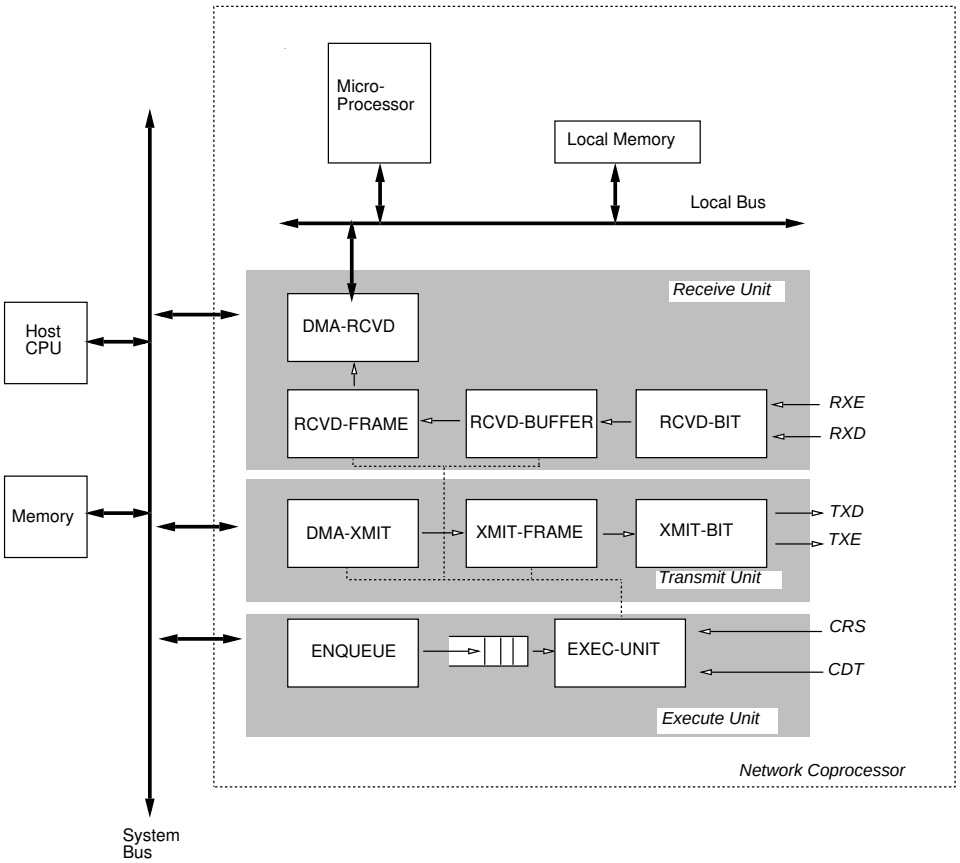


Figure 65: *Network controller block diagram*

Command	Description
start	address to store frame in global memory
stop	no parameter is needed
ctaddr	controller address on network domain
sifr	interframe spacing in bytes
jam	time in bytes jam is inserted on line
	jam inserted in the network line
preamble	number of preamble bytes
	preamble byte sent
frdelim	end of frame byte
	start of frame byte

Table 10: *Network controller instruction set*

8.2.3 Controller architecture

The controller architecture is modeled after the target system architecture shown in Section 1.7.1. A modification is addition of a local memory and local bus in order to reduce the system bus bandwidth. The controller can be thought of logically consisting of following functional units: execute, transmit and the receive unit. The network controller block diagram is shown in Figure 65.

The **Execute unit** provides for fetching and decoding of controller instructions. It provides a repertoire of eight instructions listed in Table 10. The **Receive unit** receives frames and stores them into memory. The host cpu sets aside an adequate amount of buffer space and then enables the controller. Once enabled, frames arrived asynchronously. The controller must always be ready to receive the data and store them into a free memory area. The controller checks each received frame for an address match. If a match occurs, it stores the destination and source address and length field in the next available free space. Once an entire frame is received without errors, the controller does the following:

- updates the actual count of the frames received
- fetches address of the next free receive buffer
- interrupts the cpu

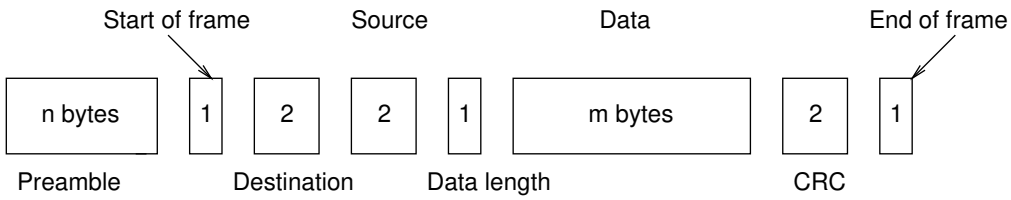


Figure 66: *Format of an ethernet frame*

Given a pointer to the memory, the **Transmit unit** generates the preamble start frame delimiter, fetches the destination address and length field from the transmit command, inserts its unique address as the source address, fetches data field from buffers pointed by the transmit command, computes and appends CRC at the end of the frame.

Figure 66 shows the format of the transmit frame. After sending a frame, the transmission unit waits some time until it starts the transmission of another frame. This interframe spacing is set by the command SIFR.

The important rate and timing constraints on the controller design are: the maximum input/output bit rate is 10 Mb/sec; maximum propagation delay is 46.4 μs; maximum jam time is 4.8 μs and the minimum inter-frame spacing is 67.2 μs.

8.2.4 Network controller implementation results

The network controller is modularly described as a set of 13 concurrently executing processes which interact with each other by means of 24 send and 40 receive operations. The total *HardwareC* description consists of 1036 lines of code.

A mixed implementation following the approach outlined in Section 6.5 was attempted by describing the software component as a single program using case descriptions. Table 11 shows the results of synthesis of application-specific hardware component of the system implementations that was synthesized in the Olympus Synthesis System and mapped using LSI logic 10K library of gates. Table 12 shows synthesis results using ACTEL library of gates. The software component is implemented in a single program containing case switches corresponding to 17 synchronization points as described in Section 5.7.2. With reference to Figure 65, the software component consists of the execution

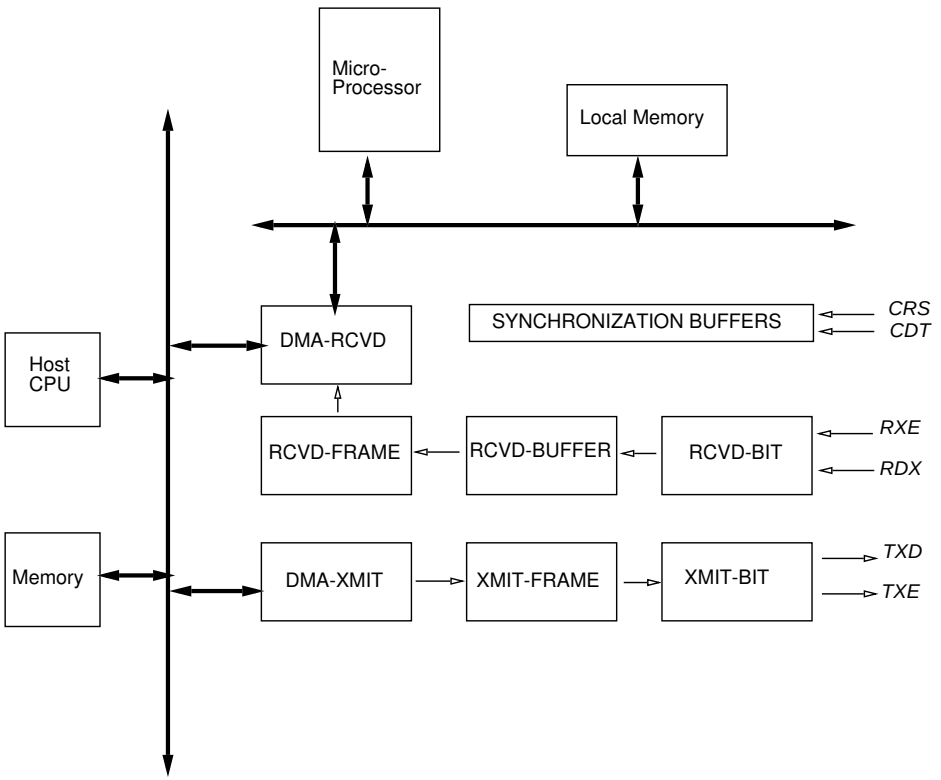


Figure 67: *Network controller implementation*

unit and portions of the `DMA_rcvd` and `DMA_xmit` blocks. The reception and transmission of data on the network line is handled by the application-specific hardware running at 20 MHz. The total interface buffer cost is 314 bits of memory elements. Table 13 lists statistics on the code generated by existing software compilers for the network controller software component implementation.

By contrast, a purely hardware implementation of the network controller requires 10882 gates (using LSI 10K library). Thus by a mixed hardware-software implementation, we are able to achieve a 20 MHz controller operation while decreasing the overall hardware cost to only one application-specific chip (or 23% in terms of gate count). The reprogrammability of software components makes it possible to increase the controller functionality, for example addition of self-test and diagnostic features, with little or no

Unit	*Process*	*Area*	*Delay*
Transmit Unit	xmit_bit	271	14.31 ns
	xmit_frame	3183	37.15 ns
	DMA_xmit	2560	45.06 ns
Receive Unit	DMA_rcvd	400	27.51 ns
	rcvd_bit	282	12.30 ns
	rcvd_buffer	127	22.09 ns
	rcvd_frame	1571	38.12 ns
Controller		8394	45.06 ns

Table 11: *Network controller synthesis results using LSI library gates*

Unit	*Process*	*Area*	*Delay*
Transmit Unit	xmit_bit	268	128.10 ns
	xmit_frame	2548	246.0 ns
	DMA_xmit	2028	472.85 ns
Receive Unit	DMA_rcvd	563	236.65 ns
	rcvd_bit	211	115.50 ns
	rcvd_buffer	121	199.28 ns
	rcvd_frame	1226	298.40 ns
Controller		7022	472.85 ns

Table 12: *Network controller synthesis results using Actel gates*

Target Processor	*Pgm & Data Size*	*Max Delay*
R3000, 10 MHz	8572 bytes	56 cycles, 5.6 μs
8086, 10 MHz	1295 bytes	115 cycles, 11.5 μs

Table 13: *Network controller software component*

increase in dedicated hardware required. Finally, we note that a purely software implementation of the controller would allow for a maximum of 2 cycles for sampling the input bit stream, and therefore, is quite unlikely to meet the performance constraints.

Figure 68 shows a results of simulation of the final network controller design. This simulation portion shows when the controller is being programmed for transmission. Instructions are supplied to the controller by external programmer (the host CPU) which are enqueue in a 1-deep queue, `ctqueue`. The last pulse on signal `ctqueue_ak` indicates transmission enable instruction after which data is transmitted serially through signal TXD.

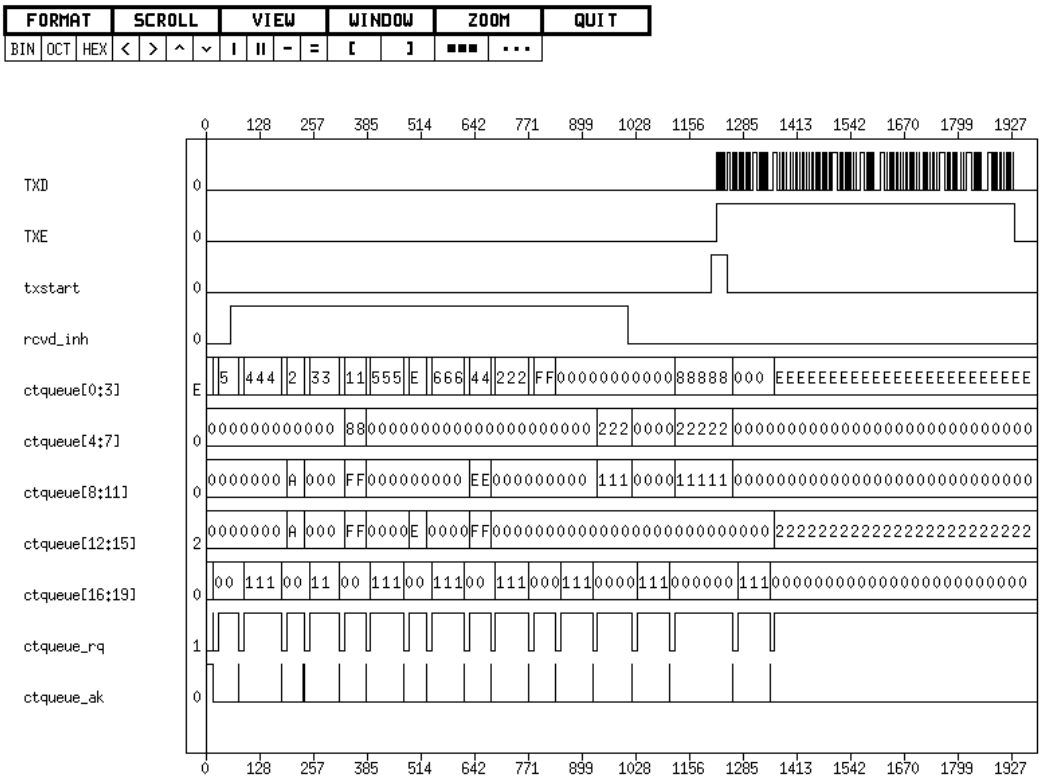


Figure 68: *Network controller simulation*

Chapter 9

Summary, Conclusions and Future Work

We have addressed the broad problem of hardware-software co-synthesis for digital systems. This formulation of the co-synthesis problem is based on the extension of the high-level synthesis techniques to system-level by generalizing the concept the resources, and treating the processor as another *resource*. This treatment of processor in a system design proves to be fundamentally different mindset than is the case in the conventional system design, where most of the system design issues revolve around utilizing the maximum performance out of the processor. However, due to the differences in the execution rate and timing of operations, the problems of software generation and its interface to the hardware are much more complicated than the problems of operation scheduling and resource allocation in high-level synthesis. Our extension of high-level synthesis approach toward system cosynthesis is nowhere more apparent than in the input language used. We start with a description of system functionality in a hardware description language (HDL). This choice of HDL is made for two primarily practical reasons. One, it provides us a means of comparison to an existing path from purely hardware implementations starting from the same input. Two, it constrains the scope of input description well enough so that a simple graph based model can be used to abstract this specification on which systematic analysis and transformations needed for cosynthesis can be developed. But this choice of a hardware description language is far from being ideal. The chief

limitation being use of extensive control flow structure necessary to describe the functionality in an algorithmic manner. These control flow structures ultimately translate into an hierarchical organization of graph models, that is not easy to alter. In particular, these structures strongly influence the system partitioning and program thread generation and alternative specifications of the same system functionality lead to different cosynthesis results.

From the input description using a HDL, we develop a graph based model that is applicable to synthesis of both hardware and software due to its explicit treatment of operation-level concurrency and synchronization. The graph model is devised to support implementations of graph models that execute at very different speeds by means of message-passing based communications between models. The absence of any shared memory between different process models obviates the need for lock-step executions of separate graph models. At the same time, the operations within a graph model communicate by means of shared memory, providing a way for efficient individual hardware or software implementations. Through this dichotomy of communication implementation, a hardware-software system is described at the level of individual graphs as being implemented in either hardware or software.

Based on this graph based model, the problem of cosynthesis is broken into subproblems of performance modeling and estimation for hardware and software, the identification of hardware and software and finally the synthesis and integration of hardware and software components. Identification of hardware and software is based on an analysis of the timing constraints. The timing constraints are of two types: minimum and maximum delay constraints between time of execution of pairs of operations and upper and lower bounds on the rate of execution of an operation. In conventional terms, the min/max delay constraints are ‘latency-type’ constraints, whereas the execution rate constraints are ‘throughput-type’ constraints, though the definitions of latency and throughput must be clearly understood in the context of multiple rate systems modeled by the flow graphs. Constraint analysis proceeds by attempting to determine if the constraints are satisfied by an an implementation by performing graph analysis on the constraint graph model. However, such an analysis is not always conclusive. The cases when the deterministic constraint analysis fails are identified by presence of cycles containing data-dependent

loop or synchronization operations (collectively referred to as $\mathcal{ND}$ operations). A notion of marginal satisfiability of constraints is developed that determines probabilistic satisfaction of timing constraints under a specified bound on the probability of violations. This constraints analysis is made a part of the partitioning procedure in determining which flow graphs should be implemented either in hardware or in software.

Synthesis of hardware is carried out by use of high-level synthesis techniques. Though central to the task of hardware-software cosynthesis, synthesis of hardware forms a part of the previous research on Olympus synthesis systems, and is not considered in this dissertation. Synthesis of software poses challenging issues due to the need for serialization of all operations and development of a low overhead runtime system. We use a FIFO-based runtime scheduler to implement the software as a set of multiple concurrent coroutines. The overhead due to such a scheduler is characterized. Finally, the hardware-software system is put together by design of a low overhead hardware-software interface.

It is clear that research in hardware-software co-synthesis spans several disciplines from CAD-theoretic aspects of algorithms for constraint analysis and partitioning to system implementation issues of concurrency and run-time systems to support multiprogramming and synchronization. This dissertation makes one of the first attempts at developing the various sub-problems that are solved in an effort to develop an effective and practical co-synthesis approach. In the process, several simplifications are made, all in an attempt to keep the focus on essentials of the co-synthesis problem while delegating peripheral (though sometimes no less important) problems to a workable engineering solution. As a result, we are able to put together a complete co-synthesis solution for system designs that are modeled using hardware description languages. We have demonstrated the feasibility of achieving co-synthesis, thus validating the basic hypothesis of the thesis.

9.1 Future Work

Due to the broad scope of transformations needed to realize interacting hardware and software components that can execute at widely different rates, which synchronize only when necessary, what is needed is a representation of the system model that is structurally

as simple the flow graph model used here, and yet it supports ease in implementation of a variety of transformations primarily related to altering the flow of control and data. For control purposes an algebraic approach appears to be promising. For example, consider the following two pieces of code, the outer most while statement explicitly models the infinite repetition:

```
        while(1) {                          while(1) {
            if (condition)                      while (condition)
a:              A;                  a:              A;
b:          B;                      b:          B;
        }                                   }
```

While it is hard to reason about their equivalence when abstracted as graph models, it is easy to capture them algebraically, for example, as path expressions where simplifications based on axiomatic rules can be made. For the example, the control flow can be shown to be equivalent by proving the following equivalence: $(a + b)^{\omega} = (a^{*}b)^{\omega}$ where ω represents infinite repetition. However, this abstraction is not sufficient either since it completely ignores the data flow. In this context, models that provide encapsulation of both data and control flow, albeit as different levels of abstraction would find more appeal in system cosynthesis.

Hardware-software interface remains to be a key area where the need for appropriate abstractions is most keenly felt. This is perhaps because in our formulation of the cosynthesis problem the abstraction of interface takes a step backwards due to the choice of hardware description language to specify system functionality. Most HDLs either ignore the interface abstraction completely or mix the issues in interface functionality and its format, in a manner which akin to the similarity in *data types* and *data formats* in low level program languages. An unfortunate side-effect of inadequate interface abstraction is the strong dependence of the hardware functionality upon the type of interface chosen for the system design, and in most HDLs, the functionality must be completely rewritten once the system interface or the protocol(s) used to implement interface are altered. To be sure, the problem of interface abstraction at system-level is more complicated than the development of data types in programming languages, due to the fact that interface formats are intimately tied to the timing behavior of system. Thus a need exists to devise abstraction mechanisms that not only consider spatial format of data in terms of

organization and encoding of bits and words, but also temporal relationships, for example, multiplexing and synchronization relationships between data objects.

Finally, several extensions of the target architecture are possible and must be explored in order to broaden the applicability of cosynthesis to embedded systems. We have so far considered only single processor systems. However, there is no reason that multiple processor can not be used in such systems to improve performance. However, a multi-level memory model must be supported in order to efficiently implement a multiple processor target architecture.

Bibliography

[AB91] Tod Amon and Gaetano Borriello. Sizing Synchronization Queues: A Case Study in Higher Level Synthesis. In *Proceedings of the* 28th *Design Automation Conference*, pages 690–693, June 1991.

[AFR80] K. Apt, N. Francez, and WW. De Roever. A Proof System for Communicating Sequential Processes. *ACM Trans. on Programming Languages and Systems*, 27(2):359–385, July 1980.

[AS83] G. R. Andrews and F. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–44, March 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[Bad93] M. L. Bader. Market survey. *Bader Associates, Mountain View, California*, 1993.

[BCM+88] R. K. Brayton, R. Camposano, G. De Micheli, R. Otten, and J. van Eijndhoven. The Yorktown Silicon Compiler System. In Daniel Gajski, editor, *Silicon Compilation*, pages 204–310. Addison Wesley, 1988.

[BEW88] David Bustard, John Elder, and Jim Welsh. *Concurrent Program Structures*, page 3. Prentice Hall, 1988.

[BHLMar] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulations*, to appear.

[BK90] J. A. Bergstra and J. W. Klop. Applications of Process Algebra. In J. C. M. Baeten, editor, *An introduction to process algebra*, pages 1–21. Cambridge University Press, 1990.

[BL90a] F. C. Belz and D. C. Luckham. A new approach to prototyping ada-based hardware/software systems. In *Proceedings of TRI-Ada*, pages 141–155, December 1990.

[BL90b] F. C. Belz and D. C. Luckham. A new language-based approach to the rapid construction of hardware/software system prototypes. In *Proc. Third International Software for Strategic Systems Conference*, pages 8–9, February 1990.

[BRV89] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. In J. McCanny, J. McWhirter, and E. Swartzlander Jr., editors, *Systolic Array Processors*, pages 300–309. Prentice Hall, 1989.

[BRX93] E. Barros, W. Rosenstiel, and X. Xiong. Hardware/Software Partitioning with UNITY. In *Notes of Workshop on Hardware/Software Co-design*, October 1993.

[BV92] K. Buchenrieder and C. Veith. Codes: A practical concurrent design environment. In *Notes from Internation Workshop on Hardware-Software Codeign*, 1992.

[BW90a] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[BW90b] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1990.

[Cam90] Raul Camposano. Path-based scheduling for synthesis. *IEEE Transactions on CAD/ICAS*, 10(1):85–93, January 1990.

[Cer72] V. Cerf. *Multiprocessors, Semaphores and a Graph Model of Computation*. PhD thesis, UCLA, April 1972.

[CGH+93a] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni Vincentelli. A formal specification model for hardware/software codesign. Memorandum UCB/ERL M93/48, UC Berkeley, June 1993.

[CGH+93b] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni Vincentelli. Synthesis of mixed software-hardware implementations from CFSM specifications. Memorandum UCB/ERL M93/49, UC Berkeley, June 1993.

[Cha82] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 17(6):201–207, 1982.

[CK86] R. Camposano and A. Kunzmann. Considering Timing Constraints in Synthesis from a Behavioral Description. In *Proceedings of the International Conference on Computer Design*, pages 6–9, 1986.

[CKR84] R. Camposano, A. Kunzmann, and W. Rosenstiel. Automatic Data Path Synthesis from DSL Specifications. In *Proceedings of the International Conference on Computer Design*, pages 630–635, 1984.

[CM88] K. Chandy and J. Misra. *A Foundation of Parallel Programs Design*. Prentice-Hall, 1988.

[COB92] Pai Chou, Ross Ortega, and Gaetano Borriello. Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 488–495, Santa Clara, November 1992.

[Con63] M. E. Conway. Design of a Separate Transition-Diagram Compiler. *Comm. of the ACM*, 6:396–408, 1963.

[CPTR89] C. M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey. HYPER: an interactive synthesis environment for high performance real time applications. In *Proceedings of the International Conference on Computer Design*, pages 432–435, Cambridge, MA, October 1989.

[CR89] R. Camposano and W. Rosenstiel. Synthesizing Circuits from Behavioral Descriptions. *IEEE Transactions on CAD/ICAS*, 8(2):171–180, February 1989.

[CS61] D. R. Cox and Walter L. Smith. *Queues*. John Wiley and Sons, 1961.

[Das85] B. Dasarathy. Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Method of Validating Them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[Dij75] E. W. Dijkstra. Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. *CACM*, 18(8):453–457, August 1975.

[Fis91] P. A. Fishwick. Heterogeneous Decomposition and Coupling for Combined Modeling. In *1991 Winter Simulation Conference*, pages 1199–1208, 1991.

[FKD92] D. Filo, D. C. Ku, and G. De Micheli. Optimizing the control-unit through the resynchronization of operations. *INTEGRATION, the VLSI Journal*, 13:231–258, 1992.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the Design Automation Conference*, pages 175–181, 1982.

[Fos72] J. B. Fosseen. Representation of Algorithms by maximally Parallel Schemata. Thesis, EE, MIT, 1972.

[FSC73] F. N. Fritsch, R. E. Shafer, and W. P. Crowley. Solution of the transcendental equation $we^w = \chi$. *Communications of the ACM*, 16:123–124, 1973.

[GCM92a] Rajesh K. Gupta, Claudionor Coelho, and G. De Micheli. Program Implementation Schemes for Hardware-Software Systems. In *International Workshop on Hardware-Software Co-design*, October 1992.

[GCM92b] Rajesh K. Gupta, Claudionor Coelho, and G. De Micheli. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Proceedings of the* 29th*Design Automation Conference*, pages 225–230, June 1992.

[GCM94] Rajesh K. Gupta, Claudionor Coelho, and G. De Micheli. Program Implementation Schemes for Hardware-Software Systems. *IEEE Computer*, January 1994.

[GH74] Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory*. John Wiley and Sons, 1974.

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

[GM90] R. Gupta and G. De Micheli. Partitioning of Functional Models of Synchronous Digital Systems. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 216–219, Santa Clara, November 1990.

[GM91] Rajesh K. Gupta and G. De Micheli. Vulcan - A System for High-Level Partitioning of Synchronous Digital Systems. CSL Technical Report CSL-TR-471, Stanford University, April 1991.

[GM92] Rajesh K. Gupta and G. De Micheli. System-level Synthesis Using Reprogrammable Components. In *Proceedings of the European Design Automation Conference*, pages 2–7, March 1992.

[GM93] Rajesh K. Gupta and Giovanni De Micheli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[Har92] David Harel. Biting the silver bullet. *IEEE Computer*, pages 8–20, January 1992.

[HE92] J. Henkel and R. Ernst. Ein softwareorientierter Ansatz zum Hardware-Software Co-Entwurf. In *Proceedings Rechnergestuetzter Entwurf und Architektur mikroelektronischer Systeme.*, pages 267–268, Darmstadt, Germany, 1992.

[Hec93] Andre Heck. *Introduction to Maple*. Springer-Verlag, 1993.

[HHR+91] R. W. Hartenstein, A. G. Hirschbiel, M. Riedmuller, K. Schmidt, and M. Weber. A novel ASIC design approach based on a new machine paradigm. *IEEE Journal of Solid-State Circuits*, 26(7):975–989, July 1991.

[HHW89] Reiner W. Hartenstein, Alexander G. Hirschbiel, and Michael Weber. Mapping Systolic Arrays onto the Map-oriented Machine. In J. McCanny, J. McWhirter, and E. Swartzlander Jr., editors, *Systolic Array Processors*, pages 320–336. Prentice Hall, 1989.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufman, 1990.

[HS71] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the Design Automation Conference*, pages 155–163, 1971.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[IEE87] IEEE. *IEEE Standard VHDL Language Reference Manual, Std 1076*. IEEE Press, New York, 1987.

[JJ93] K. P. Juliussen and E. Juliussen. *The 6th Annual Computer Industry Almanac 1993*. The Reference Press, Austin, TX, 1993.

[JMP89] R. Jain, M. J. Mlinar, and A. Parker. Area-time model for synthesis of non-pipelined designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 48–51, November 1989.

[Joh83] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, Cambridge, Mass., 1983.

[Kin67] P. J. H. King. Decision Tables. *The Computer Journal*, 10(2), August 1967.

[KL70] Brian W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, February 1970.

[KL93] Asawaree Kalavade and Edward A. Lee. A Hardware-Software Codesign Methodology for DSP Applications. *IEEE Design and Test Magazine*, pages 16–28, September 1993.

[KLM93] Tilman Kolks, Bill Lin, and Hugo De Man. Sizing and Verification of Communication Buffers for Communicating Processes. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 660–664, November 1993.

[KM90a] D. Ku and G. De Micheli. HardwareC - A Language for Hardware Design (version 2.0). CSL Technical Report CSL-TR-90-419, Stanford University, April 1990.

[KM90b] D. Ku and G. De Micheli. High-level Synthesis and Optimization Strategies in Hercules and Hebe. In *Proceedings of the European ASIC Conference*, pages 111–120, Paris, France, May 1990.

[KM90c] D. Ku and G. De Micheli. Relative Scheduling under Timing Constraints. In *Proceedings of the 27th Design Automation Conference*, pages 59–64, Orlando, June 1990.

[KM92a] David Ku and Giovanni De Micheli. *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[KM92b] David Ku and Giovanni De Micheli. Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits. *IEEE Transactions on CAD/ICAS*, 11(6):696–718, June 1992.

[KR93] F. J. Kurdahi and C. Ramachandran. Evaluating layout area tradeoffs for high level applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(1):46–55, March 1993.

[Ku91] David C. Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications*. PhD thesis, Stanford University, June 1991.

[LHG+89] Edward A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya. Gabriel: A design environment for dsp. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, volume 37, pages 141–146, November 1989.

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

[Lue79] David G. Luenberger. *Introduction to dynamic systems: theory, models and applications*. Wiley, 1979.

[LVBA93] D. C. Luckham, J. Vera, D. Bryan, and L. Augustin. Partial Ordering of Event Sets and Their Application to Prototyping Concurrent Timed Systems. *Journal of Systems and Software*, July 1993.

[LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2, 1982.

[LW83] Y. Liao and C. Wong. An algorithm to compact a VLSI symbolic layout with mixed constraints. *Proceedings of the IEEE Transactions on CAD/ICAS*, 2(2):62–69, April 1983.

[McF78] M. C. McFarland. The Value Trace: A Data Base for Automated Digital Design. Technical Report DRC-01-4-80, Design Research Center, Carnegie-Mellon University, November 1978.

[Mea89] A. Mok and et. al. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proceedings of the Sixth IEEE Workshop Real-Time Operating Systems and Software*, pages 74–80, May 1989.

[Mic94] Giovannni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[Mil90] Robin Milner. *Handbook of Theoretical Computer Science*, chapter Operational and Algebraic Semantics of Concurrent Processes. Elsevier-Science Publishers, 1990.

[MKMT90] G. De Micheli, David C. Ku, Frederic Mailhot, and Thomas Truong. The Olympus Synthesis System for Digital Design. *IEEE Design and Test Magazine*, pages 37–53, October 1990.

[Mok83] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, M.I.T., 1983.

[Mol82] Michael K. Molloy. Performance Analysis using Stochastic Petri Nets. In *IEEE Transactions on Computers*, pages 913–917, September 1982.

[OG76] S. Owicki and D. Gries. Verifying Properties of Parallel Programs. *Communications of the ACM*, 19(5):279–285, May 1976.

[OH93] Kunle Olukotun and Rachid Helaihel. Automating architectural exploration with a fast simulator. In *Notes of the Workshop on Hardware-Software Co-design*, 1993.

[oS88] United States National Bureau of Standards. *Data encryption standard*. National Technical Information Service, 1988.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. A Program for Data Path Synthesis. In *Proceedings of the* 23rd*Design Automation Conference*, pages 461–466, June 1986.

[PS90] C. Y. Park and Alan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. In *Proceedings of the* 11th*IEEE Real-Time Systems Symposium*, pages 72–81, December 1990.

[RMV+88] J. Rabaey, H. De Man, J. Vanhoof, G. Goosens, and F. Catthoor. Cathedral II: A Synthesis System for Multiprocessor DSP Systems. In Daniel Gajski, editor, *Silicon Compilation*, pages 311–360. Addison Wesley, 1988.

[Sar89] Vivek Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. MIT Press, Cambridge, Mass., 1989.

[SB88] M. E. Smid and D. K. Branstad. Data encryption standard: past and future. *Proceedings of the IEEE*, 76(5):550–559, May 1988.

[SB91] M. B. Srivastava and R. W. Broderson. Rapid-Prototyping of Hardware and Software in a Unified Framework. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 152–155, Santa Clara, 1991.

[Sch90] H. D. Schwetman. Introduction to process-oriented simulation and csim. In *Proc. Winter Simulation Conference*, New Orleans, LA, December 1990.

[Sch92] David Scholefield. The Formal Development of Real-Time Systems. Technical report, (ch. 4), University of York, York, Heslington, February 1992.

[Sha79] S. D. Shapiro. A stochastic Petri net with applications to modeling occupancy times for concurrent task systems. *Networks*, 9:375–379, 1979.

[Sha86] Moe Shahdad. An overiview of vhdl language and technology. In *Proceedings of the Design Automation Conference*, pages 320–326, July 1986.

[Sha89] A. Shaw. Reasoning about Time in Higher Level Language Software. *IEEE Trans. Software Engg.*, 15(7):875–889, July 1989.

[SJR+91] C. S. Shung, R. Jain, K. Rimey, E. Wang, M. B. Srivastava, E. Lettang, S. K. Azim, P. N. Hilfinger, J. Rabaey, and R. W. Broderson. An integrated CAD system for algorithm-specific IC design. *IEEE Transactions on CAD/ICAS*, pages 447–463, April 1991.

[SR88] John A. Stankovic and K. Ramamritham. *Hard real-time systems*. Computer Society Press, 1988.

[Sri92] M. B. Srivastava. *Rapid-Prototyping of Hardware and Software in a Unified Framework*. PhD thesis, UC Berkeley (and Memorandum No. UCB/ERL M92/67), June 1992.

[SSM+92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, 1992.

[SSR89] Roberto Saracco, J. R. W. Smith, and Rick Reed. *Telecommunications systems engineering using SDL*. North-Holland, 1989.

[Tar81] R. E. Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[tHM93] K. ten Hagen and H. Meyr. Timed and untimed hardware/software co-simulation: Application and efficient implementation. In *Internation workshop on hardware-software codesign*, 1993.

[TLW+90] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn. *Algorithmic and Register-Transfer Level: The System Architect's Workbench*. Kluwer Academic Publishers, 1990.

[TM91] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer, 1991.

[TRS+89] L. Thon, K. Rimey, L. Svensson, Paul Hilfinger, and R. W. Broderson. From C to Silicon with LagerIV. In *IEEE/ACM Physical Design Workshop*, May 1989.

[Wal90] S. Walters. Reprogrammable hardware emulation automates system-level ASIC validation. In *WESCON/90 Conference Record*, pages 650–653, Anaheim, California, November 1990.

[Wat84] I. Watson. Architecture and performance (fundamentals of dataflow). In F. B. Chambers *et. al.*, editor, *Distributed Computing*, pages 21–32. Academic Press, 1984.

[WHPZ87] L. T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. Ssim: A software levelized compiled-code simulator. In *Proceedings of the Design Automation Conference*, pages 2–8, 1987.

[WTH+92] Wayne Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu. The Princeton University Behavioral Synthesis System. In *Proceedings of the* 29th *Design Automation Conference*, pages 182–187, June 1992.

Appendix A

A Note on HardwareC

HardwareC was developed by Ku and De Micheli [KM90a] as an input language for specification for synchronous digital circuits. *HardwareC* follows much of the syntax and semantics of the programming language, C with modifications necessary for correct and unambiguous hardware modeling. Like C, the primitive operation in *HardwareC* consists of an assignment operation with a procedural call being the means of abstraction of sub-specifications. Procedural calls correspond to modular specification of different components of the hardware. No recursive calls of any form are allowed.

A *HardwareC* specification consists of *blocks* of statements which are identified by enclosing parentheses. The blocks are structured so that no two blocks are overlapped partially. That is, given any two blocks, they are either disjoint or one block is contained by the other block. Like C, no nested procedure declarations are allowed. Therefore, any variable that is non-local to any procedure is non-local to all procedures. Local variables are scoped *lexically* with the inner-most nested rule for structured blocks.

The basic entity for specifying system behavior is a *process*. A process executes concurrently with other processes in the system specification. A process restarts itself on completion of the last operation in the process body. Thus, there exists an implied outer-most loop that contains the body of the process. In other languages, this loop can be specified by an explicit outer loop statement. Operations within a process body need not be executed sequentially (as is the case in a process specification in VHDL, for example). A process body can be specified with varying degrees of parallelism such as

parallel (<>), data-parallel ({}) or sequential ([]).

In addition, the HDL uses specification of *declarative* model calls as blocks that describe physical connections and structural relationships between constituent processes. For hardware modeling purposes, both timing and resources constraints are allowed in the input specifications. Timing constraints are specified as min/max delay attributes between labeled statements whereas resource constraints are specified as user-specified bindings of process and procedure calls to specific hardware model instances.

Timing semantics. It is assumed that operations are performed synchronously using a global clock with a fixed cycle time. Accordingly, loop and procedure calls are assumed to be synchronous operations. There is no explicit delay associated with individual assignment statements (except in case of explicit register/port load operations as mentioned later). An assignment operation may take zero or non-zero delay time. This actual delay depends upon the delay characteristics of the resource(s) used to implement specified operation(s).

In case of multiple assignments to the same variable there are two possible interpretations:

1. the last assignment or

2. an assignment after some delay

. Resolution of which interpretation is to be used is performed by a reference stack algorithm [KM92a]. This algorithm performs variable propagation by instantiating values of the variables in the right-hand side of the assignments. In the case of variables that must be used before being reassigned a new value, second interpretation is adopted where 'some delay' corresponds to delay of 'at least' one cycle time. In addition, this interpretation can also be enforced on some assignments regardless of whether the assignment is referenced or not, by use of an explicit 'load' prefix that assigns a delay of precisely one cycle time to the respective assignment operation.

Appendix B

Bilogic Graphs

Bilogic flow graphs were introduced in Chapter 3. These graphs are similar control graphs [Cer72] which direct the flow of control in the following three ways:

1. Sequencing by means of directed edges between vertices;
2. Concurrent branching and merge is achieved by means of a conjoined fork or merge;
3. Conditional branching and merge is achieved by means of a disjoined fork and merge.

In a bilogic graph all vertices can have multiple fanin and fanout edges which are either conjoined or disjoined. In this thesis we concern ourselves with well-formed bilogic graphs, that is, graphs where a forks/merge is either conjoined or disjoined but not both. Bilogic flow graphs can be made well-formed by introduction of additional fork and merge vertices.

Theorem B.1 *Given a bilogic graph, $G_{bilogic}$, let $G_{unilogic}$ be a graph created by treating all fanin and fanout edges to be only conjoined. Then,*

$$\ell_M(G_{bilogic}) = \ell(G_{unilogic}) \tag{B.81}$$

Proof: For a given bilogic graph, $G_{bilogic}$, is length vector is defined as an expression over scalars representing fixed delays associated with operations

in the graph. This expression is constructed using the composition rules described earlier. We show by induction, that any disjoined composition in the expression can be replaced by a conjoined expression without altering the maximum value of the expression.

For the base case, when two scalars are disjoined, the maximum refers to the largest of the two scalars elements, which is also the largest element in case of conjoined composition of two scalars. Next, at any step, let us suppose that the maximum over subexpressions $\underline{\ell}_1$ and $\underline{\ell}_2$ is same for both unilogic and bilogic graphs, then the maximum over their composition is expressed as:

$$\begin{aligned} & \max(\underline{\ell}_1 \oplus \underline{\ell}_2) &=& \max(\ \ \ell_1[i], \ell_2[j]), \ \ i = 1 \ldots |\underline{\ell}_1|, \ j = 1 \ldots |\underline{\ell}_2| \\ \Rightarrow \quad & &=& \max(\ \ \max(\ell_1[i], \ell_2[j])) \\ \Rightarrow \quad & &=& \max(\underline{\ell}_1 \otimes \underline{\ell}_2) \end{aligned}$$

Thus, the maximum of path length over the composition is identical for both disjoined and conjoined compositions. Therefore, by induction the maximum over any expression using conjoined and disjoined operators is same when the disjoined operators are replaced by conjoined operators, that is,

$$\max \underline{\ell}(G_{bilogic} = \max \underline{\ell}(G_{bilogic})|_{\oplus \to \otimes} = \max \underline{\ell}(G_{unilogic})$$

♯

Bilogic graphs are series-parallel graphs. Based on this property, the following outlines the procedure for computing the paths lengths.

compute-length(v)
{
    switch |succ(v)|
        case 0: return $\underline{\delta}(v)$; break
        case 1: return $\underline{\delta}(v) \odot \underline{\ell}(succ(v))$; break;
        case $\geq$2: switch fork-type
            case conjoined; return $\underline{\delta}(v) \odot [\bigotimes_{w \in succ(v)} \underline{\ell}(w)]$; break;
            case disjoined; return $\underline{\delta}(v) \odot [\bigoplus w \in succ(v) \underline{\ell}(w)]$; break;
}

$\odot(\underline{\ell}_1, \underline{\ell}_2)$
{
    k = 1
    for $i = 1 \ldots |\underline{\ell}_1|$ do
        for $j = 1 \ldots |\underline{\ell}_2|$ do
            $d_k = \ell_1[i] + \ell_2[j]$;
            $k = k + 1$;
    return($\underline{d}$)
}

$\oplus(\underline{\ell}_1, \underline{\ell}_2)$
{   for $i = 1 \ldots |\underline{\ell}_1|$ do
        $d_i = \ell[i]$;
    for $j = 1 \ldots |\underline{\ell}_2|$ do
        $d_{i+j} = \ell[j]$;
    return($\underline{d}$)
}

$\otimes(\underline{\ell}_1, \underline{\ell}_2)$
    k = 1
    for $i = 1 \ldots |\underline{\ell}_1|$ do
        for $j = 1 \ldots |\underline{\ell}_2|$ do
            $d_k = \max(\ell_1[i], \ell_2[j])$;
            $k = k + 1$;
    return($\underline{d}$)
}

Appendix C

Processor Characterization in Vulcan

The follows the syntax of the CPU characteristics file with typical values for the DLX microprocessor indicated by comments. Comments begin with symbol '#'.

```
.cpumodel <processor_name> ;

        .cycle_time <num> ns ; # 40 ns
        .load <num> cycles ; # 2

        .address [<str>]* ; # a0 a1 ...
        .data [<str>]* ; # d0 d1 ...
        .interrupt [<str>]* ; # int0 int1 ..
        .reset <str> ; # RESET

        .max_gpr <num> ; # 31

        .bus_model ;
                .address_size <num> ; # 32
                .data_size <num> ; # 32
                .type [<muxed> , <de_muxed>] ; # de_muxed

                .de_muxed ;
                        .mem_read <str> ; # rd
                        .mem_write <str> ; # wr
                        .io_read <str> ; # iord
                        .io_write <str> ; # iowrite
                .end_de_muxed ;

                .muxed ;
                        .read <str> ;
                        .write <str> ;
                        .io <str> ;
                        .mem <str> ;
                .end_de_muxed ;

                .bus_hold <str> ;
                .bus_ack <str> ;

        .end_bus_model ;

        .timing_model ;
```

```
                # timing model

                .read_access <num> cycles ; # 1
                .write_access <num> cycles ; # 1

                .load <num> cycles ; # 2, Note CPI = 1.4 cycles.
                .store <num> cycles ; # 2

                .move <num> cycles ; # 1
                .xchange <num> cycles ; # 1

                .alu <num> cycles ; # 1
                .mpy <num> cycles ; # 6
                .div <num> cycles ; # 24
                .comp <num> cycles ; # 1

                .call <num> cycles ; # 1, Note CPI = 1.2 cycles.
                .jump <num> cycles ; # 1
                .branch <num> cycles ; # 1
                .bc_true <num> cycles ; # 2
                .bc_false <num> cycles ; # 1, CPI = 1.5 cycles.
                .return <num> cycles ; # 1

                # interrupts are all fixed target locations
                .seti <num> cycles ; # 1
                .cli <num> cycles ; # 1
                .int_response <num> cycles ; # 10 cycles

                .halt <num> cycles ; # 10 cycles

                # EA calculation delays
                .address_modes ;

                        .immediate <num> cycles ; # 0
                        .register <num> cycles ; # 0
                        .direct <num> cycles ; # 1
                        .reg_indirect <num> cycles ; # 1
                        .mem_indirect <num> cycles ;
                        .indexed <num> cycles ;
                        .other <num> cycles ; # 10

                .end_address_modes ;
        .end_timing_model ;
.endcpumodel ;
```

Appendix D

Runtime Scheduler Routines

This appendix lists the routines used to implement the context switch or the "`transfer_to`" function in the runtime scheduler. We consider two implementations: hardware implementation of control FIFO, and software implementation of control FIFO.

Hardware Control FIFO. The FIFO buffer and the associated control logic are synthesized in hardware. This leads to a simple runtime scheduler.

```
.global _transfer_to
        ;; void transfer_to (newroutine)
        ;; int newroutine;
_transfer_to:
        ;; lastPC[current] = r31;
        lhi r3,(_current>>16)&0xffff
        addui r3,r3,(_current&0xffff)
        add r6,r0,r3
        lw r3,0(r3)
        lhi r4,(_lastPC>>16)&0xffff
        addui r4,r4,(_lastPC&0xffff)
        add r7,r0,r4
        slli r3,r3,#2
        add r3,r4,r3
        sw 0(r3),r31
        ;; r31 = lastPC[newroutine];
        lw r3,0(r14)
        slli r3,r3,#2
        add r3,r7,r3
        lw r31,0(r3)
        ;; current = newroutine;
```

```
        lw r5,0(r14)
        sw 0(r6),r5
        jr r31
        nop
```

Software Control FIFO. The FIFO buffer and control are implemented in software. A data transfer from hardware to software is facilitated by means of an interrupt operation.

```
        .align 4
.global _transfer_to
        ;; void transfer_to (newroutine)
        ;; int newroutine;
_transfer_to:
        cli
        ;; lastPC[current] = r31;
        lhi r3,(_current>>16)&0xffff
        addui r3,r3,(_current&0xffff)
        add r6,r0,r3
        lw r3,0(r3)
        lhi r4,(_lastPC>>16)&0xffff
        addui r4,r4,(_lastPC&0xffff)
        add r7,r0,r4
        slli r3,r3,#2
        add r3,r4,r3
        sw 0(r3),r31
        ;; r31 = lastPC[newroutine];
        lw r3,0(r14)
        slli r3,r3,#2
        add r3,r7,r3
        lw r31,0(r3)
        ;; current = newroutine;
        lw r5,0(r14)
        sw 0(r6),r5
        movi2s r31
        rfe
        nop

        .align 4
.global _int1
_int1:
        sw -4(r14),r3
        sw -8(r14),r4
        sw -12(r14),r5

        ; *int1_ak = 0
        lhi r3,0x00b0
```

```
        addui r3,r3,0
        sw 0(r3),r0

        ; empty = 0
        lhi r3,(_empty>>16)&0xffff
        addui r3,r3,(_empty&0xffff)
        sw 0(r3),r0

        ; queuein = (queuein + 1) & 0xf
        lhi r5,(_queuein>>16)&0xffff
        addui r5,r5,(_queuein&0xffff)
        lw r3,0(r5)
        add r3,r3,#1
        and r3,r3,#15
        sw 0(r5),r3

        add r5,r0,r3

        ; controlfifo[queuein] = 1
        lhi r4,(_controlfifo>>16)&0xffff
        addui r4,r4,(_controlfifo&0xffff)
        slli r3,r3,#2
        add r3,r4,r3
        addi r4,r0,#1
        sw 0(r3),r4


        ; full = (queuein == queueout)
        lhi r4,(_queueout>>16)&0xffff
        addui r4,r4,(_queueout&0xffff)
        lw r4,0(r4)
        seq     r5,r5,r4

        lhi r3,(_full>>16)&0xffff
        addui r3,r3,(_full&0xffff)
        sw 0(r3),r5

        ;; Restore the saved registers
        lw r5,-12(r14)
        lw r4,-8(r14)
        lw r3,-4(r14)
        rfe
        nop

< similarly for other interrupts >
```

Appendix E

Index of Notations

Symbol	*Description*	§
N	Set of natural numbers	
Z^+	Set of positive integers	
G	Flow graph model	3.3
V	Set of vertices in graph	3.3
E	Set of edges in graph	3.3
χ	Enabling expression	3.3
v	A operation vertex	3.3
e	A directed edge	3.3
$>$	Dependency	3.3
$\succ$	Control dependency	3.3
$\prec$	Data dependency	3.3
$>^*$	Transitive dependency	3.3
Φ	System model	3.3
G^*	Hierarchy relation	3.3.2
$G^{>*}$	Transitive closure of G	3.3
ς	State of a vertex	3.3.3

Symbol	*Description*	§
s_r, s_e, s_d	Vertex state values	3.3.3
$I(G)$	Implementation of G	3.3.4
Υ	Runtime scheduler	3.3.4
S	Hardware size of an operation	3.3.4
$M(G)$	Variables used by G	3.3.4
$P(G)$	Graph model pinout	3.3.4
δ	Operation delay function	3.3.4
$\lambda(G), \lambda(T)$	Graph/Thread latency	3.3.4
ℓ	Path length	3.3.4
$\underline{\ell}$	Path length vector	3.3.4
ℓ_m	Smallest element of $\underline{\ell}$	3.3.4
ℓ_M	Largest element of $\underline{\ell}$	3.3.4
$\odot$	Sequential composition operator	3.3.4
$\otimes$	Conjoined composition operator	3.3.4
$\oplus$	Disjoined composition operator	3.3.4
$\widetilde{\rho}$	Instantaneous rate of execution	3.3.4
$\widehat{\rho}$	Discrete rate of execution	3.3.4
τ	Cycle time associated with a model	3.3.4
ρ	Rate of execution	3.3.4
ϱ	Rate of reaction	3.3.4
γ	Overhead delay	3.3.4
$\mathcal{ND}$	Non-deterministic delay operation	3.4.2
t_k	Execution start time	3.6
$l,\ u$	Min/max delay constraints	3.6
$r,\ R$	Rate constraints	3.6
r^G	Relative rate constraint	3.6
Υ	Runtime scheduler	4.1
Ω	Operation scheduler	4.1
Ω_s	Static schedule function	4.1
Ω_r	(Unilogic) relative schedule	4.1
Ω_{rb}	(Bilogic) relative schedule	4.1
θ	Offset	4.1
$\mathcal{A}$	Anchor set	4.1
$\mathcal{A}_b$	Bilogic anchor set	4.1
$\lvert\cdot\rvert_\infty$	Infinity norm	4.1
$\mathcal{CD}$	Conditional delay operation	4.1
G_T	Constraint graph	4.2
G_+	Parent graph of G	4.3
G_o	Parent process graph of G	4.3
$\overline{\gamma}$	Upper bound on overhead γ	4.3

Symbol	*Description*	§
$\Delta\ell$	$\ell_M - \ell_m$	4.3
γ_w	Overhead due to context switch	4.3
x	Loop index variable	4.3
$\mu(v)$	Mobility of operation v	4.3
p_v	Longest path from source to sink through v	4.3
$\parallel$	Multi-rate concurrency	3.5
$\natural$	Single-rate concurrency	3.5
Γ	$\mathcal{NCD}$ cycle	4.5
B_1	Blocking limit	4.5.2
B_k	Blocking limit for k-deep buffer	4.5.3
Pr{}	Probability of an event	4.6
$F_X()$	Probability distribution function of r.v. x	4.6
$f_X()$	Probability density function of r.v. x	4.6
ϵ	Acceptable probability of constraint violation	4.6
N_e	Buffer depth for exponentially distributed loop index	4.6.2
$\overline{x}, E[X]$	Expected value of r.v. x	4.6.2
$f_k(x)$	Erlangian distribution of type k	4.6.2
$M_X(t)$	Moment generating function of r.v. x	4.6.2
W	Lambert's W function	4.6.2
$\wp$	Stochastic process	4.7
p_i	Transition probability	4.7
w	State of a transition process	4.7
ρ_{ij}	State transition probability	4.7
$\mathbf{P}$	State transition matrix	4.7
$\mathbf{M}$	Markov Process	4.7
$\mho$	Event space	4.6
$\widehat{x}$	Upper bound on loop index	4.6
$\star$	Convolution function	4.7
e_k	Column vector with k^{th} entry as 1 rest 0	4.7
$\mathbf{I}_k$	$k \times k$ identity matrix	4.7
$\mathbf{0}_k$	Zero column vector of size k	4.7
$o(\omega)$	Set of terminal output nodes for ω	
$\iota(\omega)$	Set of terminal input nodes for ω	
Π	Processor cost model	5.1
τ_{op}	Execution time function	5.1
τ_{ea}	Address calculation delay function	5.1
t_m	Memory access time	5.1
t_i	Interrupt response time	5.1
ISA	Instruction set architecture	5.1
RM	ISA with register/memory operands	5.1

Symbol	*Description*	§
LS	ISA with load/store operations	5.1
MM	ISA with memory/memory operands	5.1
η	Operation delay in software	5.3.1
m_r	Number of memory read operations	5.3.1
m_w	Number of memory write operations	5.3.1
n_o	Number of assembly operations	5.3.1
η_{intr}	Synchronization delay of an operation	5.3.1
S^{Π}	Software size for processor Π	5.4
S_p^{Π}	Software program size for processor Π	5.4
S_d^{Π}	Software data size for processor Π	5.4
G^D	Flow graph with only data dependencies	5.4.2
r_r	Number of register read operations	5.4.2
r_w	Number of register write operations	5.4.2
Ξ	Spill vertex set	5.4.2
G_I	Conflict (interference) graph	5.4.2
ω	Number of *rvalues*	5.4.2
R_l	Maximum number of live variables	5.4.2
$\mathcal{C}$	Memory cost model	5.4.4
$\mathcal{P}$	Processor utilization	6.1
$\mathcal{B}$	Bus utilization	6.1
ϖ	A partition of the system model	6.1
ψ	Priority of a program thread	6.3