THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

VLSI, COMPUTER ARCHITECTURE AND DIGITAL SIGNAL PROCESSING

Consulting Editor
Jonathan Allen

Other books in the series:

AUTOMATIC SPEECH AND SPEAKER RECOGNITION: Advanced Topics, Chin-Hui Lee, Kuldip K. Paliwal
ISBN: 0-7923-9706-1
BINARY DECISION DIAGRAMS AND APPLICATIONS FOR VLSI CAD, Shin-ichi Minato
ISBN: 0-7923-9652-9
ROBUSTNESS IN AUTOMATIC SPEECH RECOGNITION, Jean-Claude Junqua, Jean-Paul Haton
ISBN: 0-7923-9646-4
HIGH-PERFORMANCE DIGITAL VLSI CIRCUIT DESIGN, Richard X. Gu, Khaled M. Sharaf, Mohamed I. Elmasry
ISBN: 0-7923-9641-3
LOW POWER DESIGN METHODOLOGIES, Jan M. Rabaey, Massoud Pedram
ISBN: 0-7923-9630-8
MODERN METHODS OF SPEECH PROCESSING, Ravi P. Ramachandran
ISBN: 0-7923-9607-3
LOGIC SYNTHESIS FOR FIELD-PROGRAMMABLE GATE ARRAYS, Rajeev Murgai, Robert K. Brayton
ISBN: 0-7923-9596-4
CODE GENERATION FOR EMBEDDED PROCESSORS, P. Marwedel, G. Goossens
ISBN: 0-7923-9577-8
DIGITAL TIMING MACROMODELING FOR VLSI DESIGN VERIFICATION, Jeong-Taek Kong, David Overhauser
ISBN: 0-7923-9580-8
DIGIT-SERIAL COMPUTATION, Richard Hartley, Keshab K. Parhi
ISBN: 0-7923-9573-5
FORMAL SEMANTICS FOR VHDL, Carlos Delgado Kloos, Peter T. Breuer
ISBN: 0-7923-9552-2
ON OPTIMAL INTERCONNECTIONS FOR VLSI, Andrew B. Kahng, Gabriel Robins
ISBN: 0-7923-9483-6
SIMULATION TECHNIQUES AND SOLUTIONS FOR MIXED-SIGNAL COUPLING IN INTEGRATED CIRCUITS, Nishath K. Verghese, Timothy J. Schmerbeck, David J. Allstot
ISBN: 0-7923-9544-1
MIXED-MODE SIMULATION AND ANALOG MULTILEVEL SIMULATION, Resve Saleh, Shyh-Jye Jou, A. Richard Newton
ISBN: 0-7923-9473-9
CAD FRAMEWORKS: Principles and Architectures, Pieter van der Wolf
ISBN: 0-7923-9501-8
PIPELINED ADAPTIVE DIGITAL FILTERS, Naresh R. Shanbhag, Keshab K. Parhi
ISBN: 0-7923-9463-1

SOFTWARE SYNTHESIS FROM DATAFLOW GRAPHS

by

Shuvra S. Battacharyya
Hitachi America, Ltd.

Praveen K. Murthy
University of California, Berkeley

Edward A. Lee
University of California, Berkeley

KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / London

Distributors for North America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

Consulting Editor: Jonathan Allen, Massachusetts Institute of Technology

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Copyright © 1996 by Kluwer Academic Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061

Printed on acid-free paper.

Printed in the United States of America

To my parents, and to Arundhati and Arpan

Shuvra S. Bhattacharyya

To my parents

Praveen K. Murthy

To Rhonda

Edward A. Lee

Table of Contents

1 Introduction.....1

1.1 Block Diagram Environments 2

1.2 Modularity and Code Generation 3

1.2.1. Compilers 4

1.2.2. Subroutine Libraries 4

1.2.3. Block Libraries 5

1.2.4. Memory Usage 5

1.3 Dataflow 6

1.3.1. Dataflow Machines 7

1.3.2. Dataflow Programming 9

1.4 Synchronous Dataflow 10

1.4.1. Computation Graphs 10

1.4.2. Infinite Execution 11

1.4.3. Scheduling 12

1.5 Generalizations to the SDF model 13

1.5.1. Boolean Dataflow 13

1.5.2. Well Behaved Dataflow 14

1.5.3. Multidimensional Dataflow 14

1.5.4. Cyclostatic Dataflow 14

1.5.5. Other Declarative Languages 16

1.6 Compilation Model 16

1.6.1. Inline Code Generation 17

1.7 Constructing Efficient Periodic Schedules 21
1.8 Related Work 23
1.8.1. Multiprocessor Scheduling 23
1.8.2. Compiler Transformations 25
1.8.3. Minimum Activation Schedules 26
2 Terminology and Notation 29
2.1 Graph Concepts 30
2.2 Computational Complexity 33
3 Synchronous dataflow 37
3.1 Computing the Repetitions Vector 48
3.2 Constructing a Valid Schedule 49
3.3 Scheduling to Minimize Buffer Usage 49
3.3.1. NP Completeness 52
3.3.2. Heuristic for Minimum Buffer Scheduling 54
4 Looped Schedules 57
4.1 Looped Schedule Terminology and Notation 57
4.2 Buffering Model 60
4.3 Clustering SDF Subgraphs 62
4.3.1. Projections 62
4.3.2. Blocking Vectors 63
4.3.3. Clustering 66
4.4 Factoring Schedule Loops 70
4.5 Reduced Single Appearance Schedules 82
4.6 Subindependence 85
4.7 Computation Graphs 91
5 Loose Interdependence Algorithms 93
5.1 Loose Interdependence Algorithms 94
5.2 Modem Example 103

5.3 Clustering in a Loose Interdependence Algorithm 105
5.4 Relation to Vectorization 113
6 Joint Code and Data Minimization 119
6.1 R-Schedules 119
6.2 The Buffer Memory Lower Bound for Single Appearance Schedules 122
6.3 Dynamic Programming Post Optimization 125
6.3.1. Well-ordered Graph: Sample Rate Conversion Example 131
6.4 Recursive Partitioning by Minimum Cuts (RPMC) 134
6.5 Non-uniform Filterbank Example 139
7 Pairwise Grouping of Adjacent Nodes 141
7.1 Proper Clustering 147
7.2 The Optimality of APGAN for a Class of Graphs 154
7.3 Examples 159
7.3.1. Tree-structured Filter Bank 159
7.3.2. Satellite Receiver 159
7.3.3. Chain Structured Graph with Irregular Rate Changes 160
8 Experiments 161
9 Open Issues 169
9.1 Tightly Interdependent Graphs 170
9.2 Buffering 171
9.3 Parallel Computation 171

Acknowledgments

The scheduling framework described in Chapter 5 has been implemented in Ptolemy, a design environment for simulation, prototyping, and software synthesis of heterogeneous systems [Buck94]. A large part of the implementation in Ptolemy was performed by Joseph Buck, a graduate student at the time and now with Synopsys Inc., and Soonhoi Ha, a post-doctoral fellow of U.C. Berkeley at the time and now a professor at Seoul National University. The implementation has been tested on several practical examples, such as the digital audio tape to compact disc sample rate conversion system, developed by Thomas M. Parks, a graduate student at U. C. Berkeley; and a QMF filter bank that was developed by Alan Peevers, who is now at Emu/Creative Systems.

We thank our colleagues Alan Kamas, and Juergen Teich for the helpful suggestions that they made while reviewing early versions of some of the material in this book. Sebastian Ritz of the Aachen University of Technology made several helpful comments while reviewing an early draft of the paper in which many of the principles in Chapters 4 and 5 were first presented. For their helpful suggestions, we also thank the anonymous reviewers of submission #SP-6606 to the IEEE Transactions on Signal Processing and submission #FORM93014 to the Journal of Formal Methods for System Design.

SOFTWARE SYNTHESIS FROM DATAFLOW GRAPHS

1

Introduction

Algorithms for digital signal processing (DSP) are often most naturally described by block diagrams in which computational blocks are interconnected by links that represent sequences of data values. Due to the emergence of low cost workstations and personal computing systems with graphics capabilities, it has become feasible for designers of signal processing systems to use graphical block diagram programming environments. As a result, there has been a proliferation of such programming environments in recent years, both from industrial sources and from research and educational institutions.

The synchronous dataflow (SDF) model, whose fundamental theories were developed by Karp and Miller in [Karp66] and by Lee and Messerschmitt [Lee87], has proven efficient for representing an important class of digital signal processing algorithms, and has been used as the basis for numerous DSP programming environments, such as those described in [Buck94, Kapl87, Lauw90, Lee89, Ohal91, Powe92, Prin92, Ritz92, Veig90]. The main property of the SDF model is that the number of data values produced and consumed by each functional component is fixed and known at compile time. This book describes techniques for compiling block diagram programs based on the SDF model into efficient object code for microprocessors and programmable digital signal processors, which are specialized microprocessors for DSP applications [Lee88b].

1.1 Block Diagram Environments

Block diagram programming of DSP systems dates back at least to the early 1960s, when a group at Bell Telephone Laboratories developed a block diagram compiler for simulating signal processing systems developed for visual and acoustic research [Kell61]. An early commercial system was the Block Oriented Simulation System (BOSS), which generated FORTRAN code [Shan87]. Covington [Covi87] presents a graphical programming environment for designing digital filters based on only two types of computational blocks — adders and constant gains. At Lincoln Laboratories, a graphical tool was developed for mapping signal processing algorithms onto a two dimensional array of Texas Instruments TMS32010 signal processors [Ziss87]. Similarly, at Carnegie-Mellon University, a hierarchical block diagram format was used to represent signal processing algorithms for compilation onto the iWarp multicomputer [Ohal91]. Currently, several graphical programming environments for DSP are also available commercially, such as the Signal Processing Worksystem, developed by Comdisco Systems, which is now the Alta Group of Cadence Design Systems [Barr91]; COSSAP, developed by Cadis and by Meyer's group at the Aachen University of Technology [Ritz92], now distributed by Synopsys; the DSP Station, developed by Mentor Graphics; and GRAPE [Lauw90], developed at Katholique Universitet of Leuven, now marketed by Eonic Systems. Several additional graphical programming and simulation environments for DSP are described in [Desm93, Kapl87, Karj88, Olso92, Rasu94, Reek92].

At the University of California at Berkeley, there has been a large effort in developing efficient and elaborate graphical design environments. This work is rooted in the BLOSIM simulation system developed by Messerschmitt [Mess84]. Further exploration with BLOSIM inspired the development of the SDF model [Lee87]; soon afterwards, Ho developed the first compiler for pure SDF semantics [Ho88b], targeted to the Motorola 56000 DSP, and this compiler formed the foundation for the Gabriel design environment [Lee89]. The successor to BLOSIM and Gabriel is Ptolemy [Buck94], an object-oriented framework for simulation, prototyping, and software synthesis of heterogeneous systems. Unlike Gabriel, which is based on a single model of computation — the SDF model — Ptolemy allows a system to consist of multiple subsystems that are specified with different models of computation, and Ptolemy allows the user to define new models of computation and to interface a newly-defined model with the existing models. For example, dynamic dataflow, discrete-event, and communicating processes are

some of the models of computation that are supported by Ptolemy in addition to SDF. The Ptolemy framework together with a block diagram programming interface have been used to develop DSP simulation capabilities [Buck91], as well as compilers for the Motorola 56000 DSP [Pino94] and the Sproc multiprocessor DSP from Star Semiconductor Corporation [Murt93].

As mentioned above, a primary attraction of graphical programming environments for DSP is that DSP algorithms are often most naturally represented as hierarchies of block diagrams. This attraction is not surprising given that graphical user interfaces have proved successful in several other application areas; for example, schematic capture tools for circuit design such as Viewlogic, graphical file system browsers such as Mac Finder (for Apple Macintosh computers), "what you see is what you get" (WYSIWYG) word processors, such as FrameMaker by Adobe Software, and, of-course, multimedia web browsers for the Internet.

1.2 Modularity and Code Generation

Two additional attractions of graphical environments for DSP are the support for software reuse (*modularity*) and the support for efficient compilation. Graphical programming environments for DSP normally contain palettes of graphical icons that correspond to predefined computational blocks, and the program is constructed by selecting blocks from these palettes and specifying interconnections. If some functionality is desired that is not available in the existing library, then it is usually easy to define a new function and add it to the library, upon which the new function can become available to all other users of the system. Thus, the format of graphical programming environments makes it natural and convenient to recycle software and the development effort. For example, since each function is defined only once, it becomes economical to spend a large effort to hand-optimize frequently used functions for efficiency.

An alternative means of attaining modularity that has been explored in DSP design environments is the use of libraries of subroutines that can be called from high level language programs [Egol93, Tow88]. Here, once the library is in place, the programmer has the convenience of programming in a moderately high level language, such as C or FORTRAN, while exploiting the efficiency of hand-optimized functions written in assembly language.

1.2.1. Compilers

There have been widespread reports on the inability of high-level language compilers to deliver satisfactory code for time-critical DSP applications [Geni89, Tow88, Yu93]. The throughput requirements of such applications are often severe, and designers typically resort to careful manual fine-tuning to sufficiently exploit the parallel and deeply pipelined architectures of programmable digital signal processors while meeting their stringent memory constraints. For example, a study was done by Zivojnovic *et al.* on the performance of C compilers for several popular DSPs [Zivo95]. The best performance exhibited by these compilers — by the Tartan compiler for the Texas Instruments C40, and the ADI 2.0 compiler for the Analog Devices 21060 — exhibited overheads over handwritten code of 290% and 219% for execution time, 44% and 57% for program memory, and 0% and 0% for data memory respectively. These numbers were measured on an FIR filter benchmark. The worst performance exhibited by these compilers — by the ADI 5.1 for the Analog Devices 2101 — had overheads of 775%, 250%, and 0% for execution time, program memory, and data memory respectively.

1.2.2. Subroutine Libraries

The use of optimized subroutine libraries, as described earlier, is one approach to improving efficiency without forcing the user to write or fine-tune code at the assembly language level. A second approach is to add extensions to a high level language that facilitate the expression and optimization of common signal processing operations [Lear90]. This can be highly successful in some compilers; for example, when DSP extensions to C are used, the Tartan compiler achieves an overhead of only 5% in execution time, and has no overhead in program or data memory on the FIR filter benchmark. However, in other compilers, this may not be as useful. The afore-mentioned ADI 5.1 compiler for the 2101 actually gives worse results when compiling the FIR benchmark with DSP extensions: overheads of 885%, 283%, and 0% for execution time, program, and data memory respectively [Zivo95].

Another approach is the application of artificial intelligence techniques to confer optimization expertise to high level language compilers [Yu93]. Although it has not been extensively evaluated yet, preliminary results on this method show promise.

1.2.3. Block Libraries

The alternative that is pursued in this book is the use of graphical or textual block diagram languages based on the SDF model in conjunction with hand-optimized block libraries. As is discussed in Chapter 3, the SDF model allows us to schedule all of the computations at compile-time and thus eliminates the run-time overhead of dynamic sequencing. This increased efficiency comes at the expense of reduced expressive power: computations that include data-dependent control constructs cannot be represented in SDF; however, SDF is suitable for a large and important class of useful applications, as the large number of SDF-based signal processing design environments suggests. Benchmarks on the Gabriel design environment [Lee89] showed that compilation from SDF block diagrams produced code that was significantly more efficient than that of existing C compilers [Ho88a], although not as efficient as hand-optimized code, and for a restricted model of SDF in which each computation produces only one data value on each output and consumes only one data value each input, the Comdisco Procoder block diagram compiler produced results that were comparable to the best hand-optimized code [Powe92]. The reason for this impressive performance is that traditional compilers apply optimizations mostly within basic blocks [Aho88], while SDF compilers have more knowledge of the control structure of the program (as mentioned above, the sequencing of SDF computations can be fixed at compile time) and can thus apply optimizations globally. Although the performance of the Comdisco Procoder is impressive, the restricted computational model to which its optimizations apply does not support systems that have multiple sample rates.

1.2.4. Memory Usage

An important problem that arises when compiling SDF programs is the minimization of memory requirements— both for code and data (intermediate results). This is a critical problem because programmable digital signal processors have very limited amounts of on-chip memory, and the speed and financial penalties for using off-chip memory are often prohibitively high for the types of applications, typically embedded systems, where these processors are used. In some processors, such as the Motorola DSP56001, one on-chip instruction and two on-chip data words can be accessed in parallel, while there is only one external memory interface. Thus, there is a speed penalty for accessing off-chip memory regardless of how fast the external memory is. Moreover, off-chip memory

typically needs to be static, increasing the system cost considerably. This book develops techniques to minimize the code size when compiling an SDF program, and combines these techniques with techniques for minimizing the amount of memory required to buffer data between computational blocks.

As will be discussed later, large sample rate changes result in an explosion of code size requirements if naive compilation techniques are used. This book develops a class of scheduling algorithms that minimizes code space requirements through the careful organization of loops in the target code. This scheduling framework provably synthesizes the most compact looping structures for a certain class of SDF graphs, and from our observations, this class appears to subsume most practical SDF graphs. Also, by modularizing different components of the scheduling framework and establishing their independence, it is shown that other scheduling objectives can be incorporated in a manner that does not conflict with the goal of code compactness. This is demonstrated for two specific additional objectives — decreasing the amount of memory required for data storage and increasing the amount of data transfers that occur through registers rather than through memory.

It should be noted that there have been significant efforts to improve the efficiency of code generated from high level language programs of DSP applications, such as those described in [Hart88, Kafk90, Liao95, Yu93], and the success of these efforts indicates that the range of applications that are adequately supported by high level language compilers is increasing. However we emphasize that the efficiency of the compiled code is not the only advantage of block diagram programming and the SDF model — block diagram environments often provide the most natural specification format for signal processing algorithms, and they promote the recycling of software, expertise, and the development effort. All of these advantages motivate the solutions developed in this book.

1.3 Dataflow

The principles of dataflow and their application to the development of computer architectures and programming languages were pioneered by Dennis [Denn75]. A central objective of the dataflow concept is to facilitate the exploitation of parallelism from a program. In dataflow, a program is represented as a directed graph, called a *dataflow graph*, in which the vertices, called *actors*, represent computations and the edges represent FIFO channels. These channels queue data values, encapsulated in objects called *tokens*, which are passed from

the output of one computation to the input of another. A key requirement of the computation corresponding to a dataflow actor is that it be *functional*; that is, each output value of an invocation of the computation is determined uniquely by the input values to that invocation (state can be handled by using self loops that contain initial tokens).

A dataflow representation of a computation differs fundamentally from a corresponding representation in a imperative language such as C or FORTRAN in that it specifies the function being computed rather than specifying a step-by-step procedure to compute it. This distinction between *definitional* approaches to programming, such as dataflow, and *operational* approaches, such as C or FORTRAN is explored in depth in [Ambl92]. A major disadvantage of operational approaches is that they leave the programmer responsible for a difficult task, namely ordering the computations to optimize the speed and memory requirements of the target implementation. Of course, the compiler can attempt to deduce the dependencies between computations from an operational specification and then reorder the computations in a more efficient way, but this endeavor is often made difficult or impossible by side effects, aliasing, or unstructured control-flow.

An important subclass of definitional approaches to programming are functional languages (for example, Haskell [Huda89] and ML [Miln90]), where computations are specified through compositions of functions. In these languages, programs can, in principle, be easily converted into equivalent dataflow representations [Acke82]. Lee and Parks have explored several more subtle relationships between functional languages and dataflow-based graphical programming frameworks [Lee95].

1.3.1. Dataflow Machines

Dennis applied the concepts of dataflow to pioneer a class of computer architectures called *dataflow computers*. Unlike conventional von Neumann computers in which the execution of instructions is controlled by a program counter, computations in a dataflow computer are driven by the availability of data. This is achieved by maintaining, at the machine level, a representation of the program as a dataflow graph, and by providing capabilities in hardware to detect which actors have sufficient data to fire, to execute the corresponding instructions, and to route the output values to the appropriate actor inputs.

There are at least two basic types of dataflow computers — *static* dataflow computers and *tagged-token* dataflow computers. The original dataflow computer architecture, the MIT Static Dataflow Architecture [Denn80], was of the static variety. In a static dataflow computer, at most one data value can be queued on an edge at one time. This restriction allows the storage for the edges to be allocated at compile-time, and it is enforced by adding feedback edges, called *acknowledgment arcs*, directed between the sink and source actors of the edges in the original dataflow graph. In the MIT Static Dataflow Computer, the dataflow graph is maintained at the machine level as a collection of *activity templates*, which correspond to actor invocations. Each activity template consists of an opcode that specifies the associated machine instruction, locations to hold the operands, and pointers to the appropriate operand slots of the activity templates that must receive the output value. Each time an instruction is executed, each activity template referenced by the associated destination address pointers is updated by the *Update Unit* to contain the new output value in the appropriate operand slot. For each activity template that it modifies, the Update Unit checks whether that last vacant operand slot has been filled, and if so, it forwards a reference to the activity template to the *Instruction Queue*. Entries in this queue are processed by the *Fetch Unit*, which looks up each corresponding activity template in the activity store, sends an operation packet to the *Execution Unit*, and resets the activity template.

Since the rate at which instructions are executed is limited mainly by the rate at which the Execution Unit performs computations and by the rate at which the Instruction Queue is filled, which in turn depends on the matching of operand values to activity templates, the problems that arise in conventional von-Neumann processors due to memory latencies and synchronization are mitigated. Rather than handling interprocessor synchronization and processor-memory synchronization by wasteful idle-waiting or by expensive context switches, data dependencies are enforced by the hardware for each individual instruction, and independent operations are automatically detected and exploited.

A major shortcoming of the static dataflow computer arises from the restriction that only one data value can be queued on an edge at a given time, which implies that multiple invocations of a given actor cannot be executed in parallel. This severely limits the parallelism that can be exploited in loops (iterative computations) and precludes executing multiple invocations of a subroutine in parallel. To overcome this shortcoming, Arvind and Nikhil at MIT [Arvi90], and Gurd *et al.* at Manchester University [Gurd85] independently developed and

explored the *tagged-token* concept, which permits an arbitrary number of invocations of the same actor to execute concurrently. In a tagged-token dataflow computer, an identifying tag is carried around with each token. This tag designates the subroutine invocation number, loop iteration number, and the instruction number. For example, in the MIT Tagged-Token Dataflow Machine, the *Waiting-Matching Unit* removes unprocessed tokens buffered in a *Token Queue*, and compares the tag of each token it removes with the tags of all tokens that are in the Waiting-Matching unit at that time. If a matching tag is not found, then the token is stored in the Waiting-Matching unit until a matching token arrives. Otherwise the matching token pair is forwarded to the *Instruction-Fetch Unit*, which accesses program memory to determine the appropriate machine instruction and constructs an operation packet consisting of the instruction and its operands. This operation packet is forwarded to the *ALU*, and simultaneously the operation is executed and the tag for the result token is computed. The result token and its tag are then combined and entered in the *Token Queue*.

Although dataflow computers succeed in attacking the problems of synchronization and memory latency, challenges remain in coping with the resource requirements of unpredictable and unbounded amounts of parallelism, and in amortizing the overhead incurred on sequential code. Thus, dataflow computer technology has not yet matured to the point of being commercially advantageous, and thus there are no commercially available dataflow computers to date, although some commercially available processors have incorporated dataflow concepts to a limited degree [Chas84, Schm91].

1.3.2. Dataflow Programming

This book does not use dataflow in the context of dataflow computers; instead, it applies the concepts of dataflow as they relate to program representation. Another aspect in which the use of dataflow in this book differs from dataflow computers is in the complexity of the actors — we use a *mixed grain* dataflow model, meaning that actors can represent operations of arbitrary complexity. In contrast, dataflow computers operate on *fine grain*, or *atomic*, dataflow graphs, where the complexity of the actors is at the level of individual machine instructions. In the SDF-based design environments to which this book applies, dataflow actors typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters. Finally, the use of dataflow in this book is limited by the granularity of

each actor: dataflow is used to describe the interaction between actors, but the functionality of each actor can be specified in any programming language, such as C [Ritz92], C++ [Buck94], LISP [Karj88], or a LISP/assembly language hybrid [Lee89]. In the latter, the high level language is used to customize assembly language code blocks according to compile-time parameters.

1.4 Synchronous Dataflow

Synchronous dataflow is a restricted version of dataflow in which the number of tokens produced (or consumed) by an actor firing on each output (or input) edge is a fixed number that is known at compile time. Each edge in an SDF graph also has a non-negative integer delay associated with it, which corresponds to the number of initial tokens on the edge. The application of the SDF model to mixed-grain dataflow programming of multirate DSP systems was pioneered by Lee and Messerschmitt in the mid 1980s [Lee87]. In this section, we informally outline important theoretical developments on the SDF model and their application to block diagram programming of DSP algorithms. These principles will be reviewed rigorously in Chapter 3, and they will form much of the theoretical basis for the remainder of the book.

1.4.1. Computation Graphs

Important foundations for the SDF model were laid by the definition and exploration of *computation graphs* by Karp and Miller roughly two decades before the development of SDF [Karp66]. The computation graph model is equivalent to SDF graphs, except that in addition to production and consumption parameters, an additional threshold parameter is associated with each edge. This threshold parameter, which must be greater than or equal to the corresponding consumption parameter, determines the minimum number of tokens that must be queued on the edge before the sink actor can be fired. Thus, an SDF graph is a computation graph in which the threshold parameter of each edge equals the number of tokens consumed from the edge per sink invocation.

Karp and Miller established that computation graphs are *determinate*, which means that each computation graph uniquely determines the sequence of data values produced on the edges in the graph; these sequences do not depend on the *schedule* of actor executions — that is, on the order in which the actors are invoked. Also, they developed topological and algebraic conditions to determine which subgraphs in a computation graph become deadlocked. Karp and Miller

were mostly interested in programs that terminate — that is, graphs that deadlock — and hence their techniques do not lead to solutions for constructing efficient infinite schedules, although the underlying concept of determinacy applies both to infinite and finite schedules. In Section 4.7, some techniques are given that enable efficient infinite schedules to be constructed for computation graphs by using techniques developed for SDF graphs.

1.4.2. Infinite Execution

In DSP applications, we are often concerned with operations that are applied repeatedly to samples in an indefinitely long sequence of input data, and thus when applying a dataflow representation, it is mandatory that we support infinite sequences of actor executions. For example, consider the block diagram program shown in Figure 1.1, which is taken from a snapshot of a session with the Ptolemy system [Buck94]. This program specifies a sample rate conversion system developed by Thomas Parks, a graduate student at U. C. Berkeley, to interface a digital audio tape (DAT) player to a compact disc (CD) player. The sample rates of CD players and DAT players are, respectively, 44.1kHz and 48kHz, and the system in Figure 1.1 shows a multistage implementation of the conversion between these rates. The sample rate conversion is performed by three polyphase FIR filters that respectively perform 3:2, 7:5 and 7:18 rate conversions, and the cascade of blocks rooted at each filter's output simply scales the corresponding signal and displays its frequency content.

The system represented in Figure 1.1 would normally receive input continuously from the DAT player. Each rate-changing FIR filter is applied repeatedly

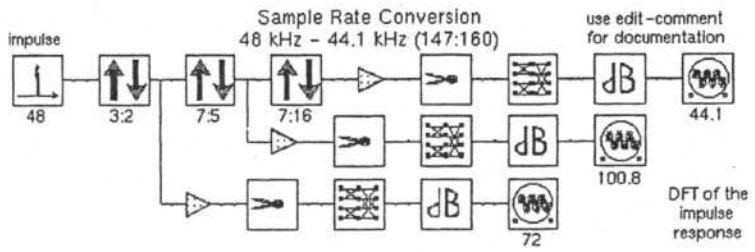


Figure 1.1. A snapshot of a session with the Ptolemy system [Buck92] that shows a sample rate conversion system for interfacing between a digital audio tape player and a compact disc player.

to successive data items that emerge from the output of the previous stage of the chain. In just 10 minutes, this system must process over 28 million input samples, and we see that it makes sense to model the input data sequence as a *semi-infinite* sequence that starts at some fixed time (the time when the system is activated) and extends to infinity. Correspondingly, we model the computation represented in Figure 1.1 as an infinite sequence of actor executions.

1.4.3. Scheduling

Three important issues emerge when attempting to derive an implementation of an infinite schedule from a dataflow graph. First, infinite schedules have the potential of requiring unbounded amounts of memory to buffer tokens as they are queued along the graph edges. Second, if deadlock arises, no more executions are possible and the infinite schedule cannot be carried out; similarly, if a subsystem becomes deadlocked, no more actors in that subsystem can be executed (even though it may be possible to continue executing actors outside the subsystem). In either case, if we are attempting to implement a system in which all operations are applied repeatedly on conceptually infinite data, then deadlock indicates an error.

Finally, we must provide a mechanism to sequence the actor executions in accordance with the given schedule. One option is to implement a software kernel that dynamically detects which actors have sufficient data on their inputs to be fired and determines when these actors are executed. However, the run-time overhead of this scheme is undesirable, particularly when a significant percentage of the invocations requires low computation time. An alternative is to store the schedule in memory as an infinite loop, thereby achieving *static scheduling*. Clearly this is only feasible if the schedule is *periodic*.

Lee and Messerschmitt resolved these issues for SDF graphs by providing efficient techniques to determine at compile-time whether or not an arbitrary SDF graph has a periodic schedule that neither deadlocks nor requires unbounded buffer sizes [Lee87]. They also defined a general and efficient framework for constructing such a periodic schedule whenever one exists. The suitability of SDF for describing a large class of useful signal processing applications and the facility for achieving the advantages of static scheduling have motivated the use of SDF and closely related models in numerous design environments for DSP [Lauw90, Lee89, Ohal91, Prin92, Ritz92, Veig90]. A large part of this book is devoted to

constructing static periodic schedules in such a way that the resulting target program is optimized.

1.5 Generalizations to the SDF model

A number of generalizations of the SDF model have been studied. In these new models, the methods for analyzing SDF graphs were extended or combined with additional techniques to incorporate actors that are more general than SDF, along with, in most cases, new techniques for constructing schedules. The objectives were to maintain at least a significant part of the compile-time predictability of SDF while broadening the range of applications that can be represented, and possibly, allowing representations that expose more optimization opportunities to a compiler.

1.5.1. Boolean Dataflow

An example is the *boolean dataflow (BDF) model*, which was defined by Lee in [Lee91] and explored further by Buck in [Buck93]. In this model, the number of data values produced or consumed by each actor is either fixed, as in SDF, or is a function of a boolean-valued token produced or consumed by the actor. Buck addresses the problem of constructing a non-null sequence of conditional actor invocations, where each actor is either invoked unconditionally or invoked conditionally based on the value of boolean tokens. This sequence should produce no net change in the number of tokens residing in the FIFO queue corresponding to each edge. Such an invocation sequence is referred to as a *complete cycle*, and clearly, if a finite complete cycle is found, it can be repeated indefinitely, and a finite bound on the amount of memory required (for buffering) can be determined at compile-time. The boolean dataflow model is *Turing-complete*, meaning that a small set of actors in this model can be used to compute any partially recursive function (put another way, any function from the integers to integers computable by a universal Turing machine can be computed by some BDF graph composed only of this small set of actors). This means that a number of key decision problems for BDF graphs become undecidable (see [Hopc79] for a discussion on undecidability and Turing-completeness), including the problem of finding finite complete cycles (and thus, the problem of determining whether a graph can be implemented in bounded memory). In contrast, the SDF model is not Turing complete since a properly constructed SDF graph can always be implemented in bounded memory. Thus, Buck presents heuristic techniques for finding finite complete cycles for BDF graphs. Whenever his techniques fail, the graph has to

be executed dynamically, although clustering techniques can often significantly reduce the number of tasks that have to be executed dynamically [Buck93].

1.5.2. Well Behaved Dataflow

Gao *et al.* have studied a programming model, called well-behaved dataflow, in which non-SDF actors are used only as part of predefined constructs [Gao92]. Of the two non-SDF constructs provided, one is a conditional construct, and the other is a looping construct in which the number of iterations can be data-dependent. This restriction on the use of more general actors guarantees that infinite schedules can be implemented with bounded memory. However, Gao's model, although more general than SDF, has significantly less expressive power than the BDF model of Buck. In particular, Gao's model is not Turing-complete (this follows from the fact that every graph in this model can be implemented in bounded memory).

1.5.3. Multidimensional Dataflow

Lee has proposed a multidimensional extension of SDF [Lee93] in which actors produce and consume n-dimensional rectangles of data, and each edge corresponds to a semi-infinite multidimensional sequence

$$\{x_{n_1, n_2, \ldots, n_m} | (0 \le n_1, n_2, \ldots, n_m < \infty)\}.$$

For example, an actor can be specified to produce a 2×3 grid consisting of six tokens each time it is invoked. Lee demonstrated that in addition to substantially improving the expressive power of the unidimensional SDF model, multidimensional SDF also exposes parallelism more effectively than unidimensional SDF.

1.5.4. Cyclostatic Dataflow

Also, in [Lauw94], Lauwereins *et al.* have proposed a useful generalization of the SDF model called *cyclo-static dataflow*. In cyclo-static dataflow, the number of tokens produced and consumed by an actor can vary between firings as long as the variations form a certain type of periodic pattern. For example, consider a *distributor* operator, which routes data received from a single input to each of two outputs in alternation (Figure 1.2). In SDF, this actor consumes two tokens and produces one token on each of its two outputs. In cyclo-static dataflow, by contrast, this operation can be represented as an actor that consumes one token on its input edge, and produces tokens according to the periodic pattern $1, 0, 1, 0, \ldots$ (one token produced on the first invocation, none on the second invocation, one

on the third invocation, and so on) on the output edge corresponding to edge (B, C), and according to complementary pattern $0, 1, 0, 1, \ldots$ on the edge corresponding to (B, D). A general cyclo-static dataflow graph can be compiled as a cyclic pattern of pure SDF graphs, and static periodic schedules can be constructed in this manner. An advantage of cyclo-static dataflow is that it can sometimes eliminate significant amounts of token traffic over corresponding SDF representations [Lauw94]. For example, in Figure 1.2(a), the periodic schedule for the SDF implementation requires 2 units of memory for the buffer on edge (A, B), while the cyclostatic implementation in Figure 1.2(b) requires only 1 unit of memory.

The techniques of this book are developed for pure (unidimensional) SDF graphs. Due to the close relation between SDF and Lee's multidimensional SDF, they can easily be extended work with multidimensional SDF. However, how the techniques are best extended to the other models described above is not obvious and calls for further investigation.

To avoid confusion, we emphasize that SDF is not by itself a programming language but a model on which a class of programming languages can be based. A library of predefined SDF actors together with a means for specifying how to connect a set of instances of these actors into an SDF graph constitutes a programming language. Augmenting the actor library with a means for defining new actors, perhaps in some other programming language, defines a more general SDF-based programming language. This book presents techniques to compile programs in any such language into efficient implementations.

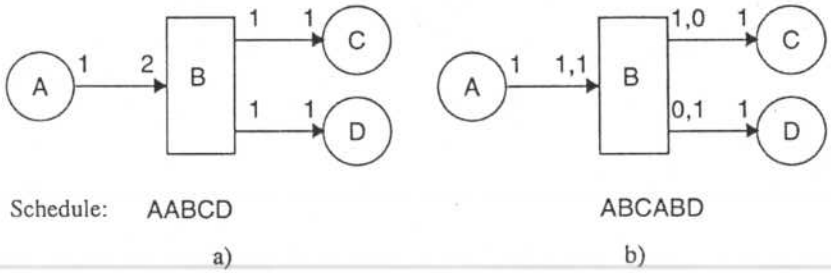


Figure 1.2. Cyclostatic dataflow compared to synchronous dataflow. Actor B is a distributor actor. a) SDF specification. b) CSDF specification.

1.5.5. Other Declarative Languages

Although the techniques in this book are presented in the context of block diagram programming, they can be applied to other DSP design environments. Many of the programming languages used for DSP, such as Lucid[Asch75], SISAL[McGr83] and Silage[Geni90] are based on or closely related to dataflow semantics. In these languages, the compiler can easily extract a view of the program as a hierarchy of dataflow graphs. A coarse level view of part of this hierarchy may reveal SDF behavior, while the local behavior of the macro-blocks involved are not SDF. Knowledge of the high-level synchrony can be used to apply "global" optimizations such as those described in this book, and the local subgraphs can be examined for finer SDF components. For example, in [Denn92], Dennis shows how recursive stream functions in SISAL-2 can be converted into SDF graphs. In signal processing, usually a significant fraction of the overall computation can be represented with SDF semantics, so it is important to recognize and exploit SDF behavior as much as possible.

1.6 Compilation Model

Figure 1.3 outlines the process of compiling an SDF block diagram program that is used in the Gabriel [Ho88a] and Ptolemy [Pino94] systems. The techniques in this book are geared towards this model. The compilation begins with an SDF representation of the block diagram program specification, and from this SDF graph, a periodic schedule is constructed. A *code generator* steps through this schedule and for each actor instance that it encounters, it generates a sequence of machine instructions, obtained from a predefined library of actor code blocks, that implements the actor. The sequence of code blocks output by the code generator is processed by a storage allocation phase that inserts the necessary instructions to route the data appropriately between actors and assigns variables to memory locations. The output of this storage allocation phase is the target program.

This form of block diagram compilation is referred to as *threading* [Bier93] since the target program is formed by linking together predefined code blocks. An alternative approach, called *synthesis*, involves translating the block diagram to an intermediate language — possibly by threading code blocks that are defined the intermediate language — and then compiling the intermediate language into C or assembly language. Examples of code generation systems that use the synthesis approach are GOSPL [Covi87], which first translates the block diagram to

LISP, and the Mentor Graphics DSP Station, which uses a language called DFL (Data Flow Language) for specification; this is Mentor's version of an applicative language called Silage [Geni90]. Most of the techniques developed in this book can be applied to synthesis; however, for clarity, the threading model is used throughout the book.

1.6.1. Inline Code Generation

In the form of threading used in this book, strictly inline code generation is performed. An alternative would be to define a subroutine for each actor and map the periodic schedule into a list of subroutine calls. However, each subroutine call induces run-time overhead. The principal components of the subroutine overhead come from saving the return address, passing arguments, allocating and deallocating local variable storage, branching to the subroutine, retrieving the return address, returning control from the subroutine, and saving and restoring the state of machine registers. Clearly if subroutines are used, the total subroutine overhead can be very detrimental if there are many actors of small granularity. The

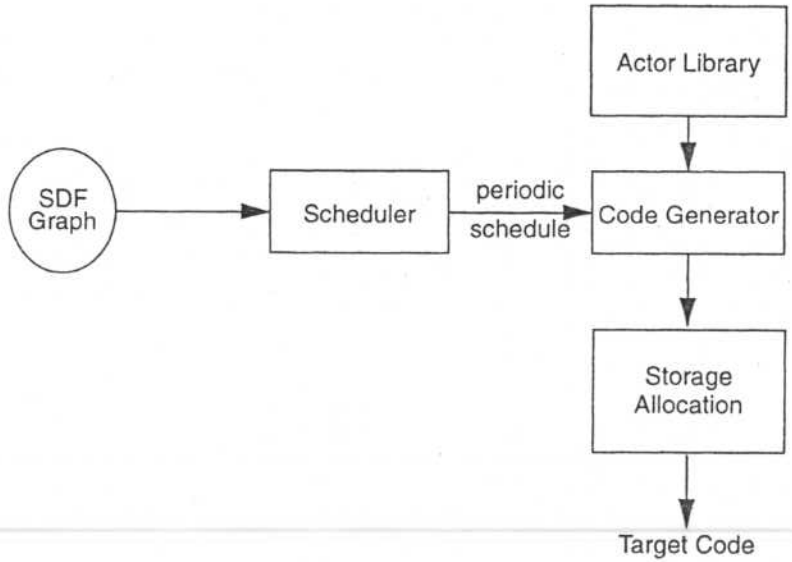


Figure 1.3. Compiling an SDF graph.

main reason that we prefer inline code over subroutines is to avoid subroutine overhead.

There is a danger, however, in using inline code, particularly for embedded system implementations, which typically can afford only very limited amounts of memory. The danger is that unmanageably large code size can result from actors that are invoked multiple times in the periodic schedule. For example, if an actor is invoked 100 times in the schedule, a straightforward inline implementation of the schedule will require 100 copies of the actor's code block to be inserted in the target code. Clearly, such code duplication can consume enormous amounts of memory, especially if complex actors having large code blocks are involved or if high invocation counts are involved.

Generally, the only mechanism to combat code size explosion while maintaining inline code is the use of loops in the target code. Clearly, if an actor's code block is encapsulated by a loop, then multiple invocations of that actor can be carried out without any code duplication. For example, for the system in Figure 1.1, as it is represented in Ptolemy, over 9000 actor code blocks are required in the target code if inline code generation is applied without employing any looping. By carefully applying loops, the target code can be reduced to only 70 code blocks. A large part of this book is devoted to the construction of efficient loop structures from SDF graphs to allow the advantages of inline code generation under stringent memory constraints. We will elaborate on this problem informally in the following section, and then present it formally in Chapter 3.

Until recently, it was widely believed that increased code size was the root cause of all aspects of the subroutine/inline code trade-off that favor the use of subroutines. However, experimental and analytical studies performed by Davidson revealed that inlining can also have a negative impact on register allocation [Davi92]. These effects however are largely artifacts of code generation conventions in modern compilers.

For example, consider the conventional *callee-save* method of maintaining the integrity of registers across subroutine calls. In this convention, the values in the registers used by a subroutine are saved (stored to memory) upon entry to the subroutine, and the saved values are restored in the corresponding registers just before returning from the subroutine. Figure 1.4 shows an example of how this convention can cause inlining to increase the amount of register-memory traffic in a program. Figure 1.4(a) shows an outline of the compiled code for two proce-


```
procedure A
     save r0
     save r1
     ...
     ...
     if (x > 0) then
          call B
     endif
     ...
     ...
     restore r0
     restore r1
```

Body of procedure A

(a)

```
procedure B
     save r2
     save r3
     ...
     body of procedure B
     ...
     restore r2
     restore r3
```

(b)

```
procedure A
     save r0
     save r1
     save r2
     save r3
     if (x > 0) then
          ...
          body of procedure B
          ...
     endif
     restore r0
     restore r1
     restore r2
     restore r3
```

Figure 1.4. An example of how inlining can increase register-memory traffic under a callee-save register save/restore convention.

dures A and B, where B is called by A. Here, x is a global variable, and the *save* and *restore* operations represent the register-memory and memory-register transfers involved in saving and restoring the registers used by a procedure. Also, we assume that B contains no subroutine calls, and the only subroutine call in A is the call to B that is shown. If procedure A is called 10 times, x is positive exactly 50% of the time, and B is not inlined in A, then it is easily verified that the calls to A result in a total of 30 register save operations and 30 restore operations. On the other hand, if B is inlined in A, as shown in Figure 1.4(b), then under the callee-save convention, the save/restore operations of B are moved to a location where they must be executed more frequently, and the 10 calls to A now result in 40 save operations and 40 restore operations.

In [Davi92] it is explained that inlining can also degrade performance with a *caller-save* convention, in which the registers used by the calling subroutine are saved by the caller just before transferring control to the callee, and the caller restores its registers just after control returns. It is also explained that the possible penalties for using inlining with the callee-save or caller-save conventions can be eliminated entirely through the application of dataflow-analysis. This has been demonstrated for callee-save systems in [Chow88] and for caller-save systems in [Davi89].

There is however one aspect of the negative interaction between inlining and register allocation that is not simply an artifact of typical compiler implementations. This is that variables of a subroutine that are placed in registers can be displaced to memory in inlined versions of the subroutine. This can lead to inefficient register allocation if frequently used variables are involved. Theoretically, this problem can be avoided since register assignments in inline code can be customized according to the context at the inlining boundaries, and thus, better register allocation is possible with inlined coded than with noninlined code. However, efficiently exploiting these opportunities for improvement is difficult, and it remains a challenge to systematically perform register allocation of inlined code in such a way that an improvement is consistently obtained over the register allocation of corresponding noninlined code [Davi92].

An important conclusion from Davidson's study is that even if the code size increase of a particular inlining application does not lead to an increase in execution time, it is not guaranteed that the inlining will not decrease performance. This refutes the prior notion that the only detrimental affects of inlining are related to increases in code size. However, Davidson's study also shows that

when the code size increase is not a factor, inlining is advantageous most of the time. The use of inline code generation is motivated by this premise that if the code size increase is tolerable, then inline code generation is *usually* more efficient than heavy use of subroutines, and it is a main purpose of this book to examine the limits to which we can exploit inline code generation under strict memory constraints when compiling SDF programs.

1.7 Constructing Efficient Periodic Schedules

This section informally outlines the interaction between the construction of periodic schedules for SDF graphs and the memory requirements of the compiled code.

To understand the problem of scheduling SDF graphs to minimize code size, it is useful to examine closely the mechanism by which iteration is specified in SDF. In an SDF graph, iteration of actors in a periodic schedule arises whenever the production and consumption parameters along an edge in the graph differ [Lee88a]. For example, consider the SDF graph in Figure 1.5(a), which contains three actors, labeled A, B and C. Each edge is annotated with the number of tokens produced on and consumed by the incident actors; for example, actor A produces two tokens each time it is invoked, while B consumes one token. The 2-to-1 mismatch on the left edge implies that within a periodic schedule, B must be invoked twice for every invocation of A. Similarly, the mismatch on the right edge implies that we must invoke C twice for every invocation of B.

Figure 1.5(b) shows four possible periodic schedules that we could use to implement Figure 1.5(a). For example, the first schedule specifies that first we are to invoke A, followed by B, followed by C, followed by B again, followed by three consecutive invocations of C. The parenthesized terms in schedules 2, 3 and 4 are used to highlight repetitive invocation patterns in these schedules. For example, the term $(2BC)$ in schedule 4 represents a loop whose iteration count is 2 and whose body is the invocation sequence BC; thus, $(2BC)$ represents the firing sequence $BCBC$. Similarly, the term $(2B(2C))$ represents the invocation sequence $BCCBCC$. Clearly, in addition to providing a convenient shorthand, these parenthesized loop terms, called *schedule loops*, present the code generator with opportunities to organize loops in the target program, and we see that schedule 2 corresponds to a nested loop, while schedules 3 and 4 correspond to cascades of loops. For example, if each schedule loop is implemented as a loop in the

target program, the code generated from schedule 4 would have the structure shown in Figure 1.5(c).

We see that if each schedule loop is converted to a loop in the target code, then each *appearance* of an actor in the schedule corresponds to a code block in the target program. Thus, since actor C appears twice in schedule 4 of Figure 1.5(b), we must duplicate the code block for C in the target program. Similarly, we see that the implementation of schedule 1, which corresponds to the same invocation sequence as schedule 4 with no looping applied, requires seven code blocks. In contrast, in schedules 2 and 3, each actor appears only once, and thus no code duplication is required across multiple invocations of the same actor. We refer to such schedules as *single appearance* schedules, and we see that neglecting the code size overhead associated with the loop control, any single appearance

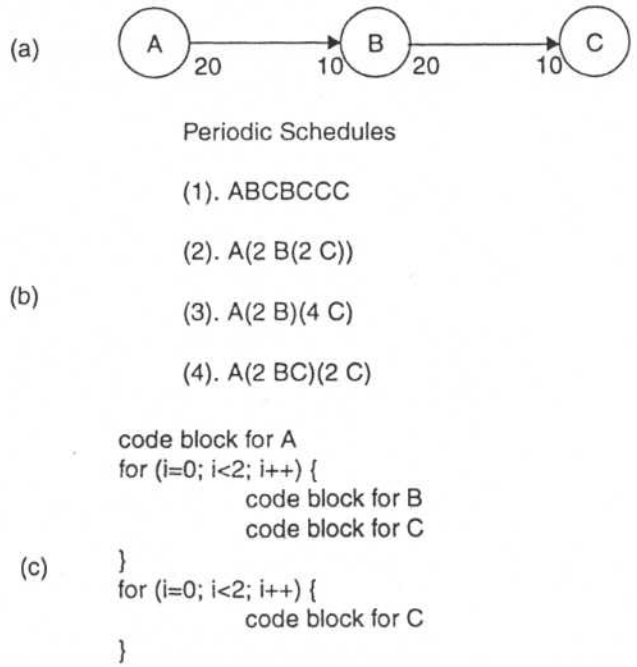


Figure 1.5. An example used to illustrate the interaction between scheduling SDF graphs and the memory requirements of the generated code.

schedule yields an optimally compact inline implementation of an SDF graph with regard to code size. Typically the loop overhead is small, particularly in many programmable DSPs, which usually have provisions to manage loop indices and perform the loop test in hardware, without explicit software control. A large part of this book is devoted to studying properties of single appearance schedules, determining when single appearance schedules exist, and systematically constructing single appearance schedules whenever they exist.

We also study the impact of scheduling on the amount of memory required to implement the buffers on the arcs in an SDF graph. For example, in Figure 1.5(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each arc, are 50, 40, 60, and 50 respectively.

1.8 Related Work

1.8.1. Multiprocessor Scheduling

Numerous research efforts have focused on constructing efficient parallel schedules from SDF graphs. These efforts mostly operate on *homogeneous SDF* graphs; that is, SDF graphs in which each actor produces a single token on each output edge and consumes a single token from each input edge. Since iteration within a periodic schedule, as defined in Subsection 1.7, does not arise in homogeneous SDF graphs, scheduling techniques for homogenous SDF graphs do not encounter the central problem addressed in this book — the management of code size and buffering when large amounts of iteration are present in general SDF graphs.

However, a number of the parallel scheduling techniques for homogeneous SDF graphs have important implications on code size. Many of these connections are related to the *unfolding factor* of a parallel schedule. The unfolding factor of a given periodic schedule S is the largest common factor (greatest common divisor) of the actor invocation counts in S (by the invocation count of an actor in S, we simply mean the number of times that the actor is invoked in S). The unfolding factor can also be viewed as the number of *minimal* periodic schedules that exist in the schedule. For example, a minimal periodic schedule for Figure 1.5(a) consists of 1 invocation of A, 2 invocations of B, and 4 invocations of C. Any schedule that invokes A, B and C U, $2U$ and $4U$ times, respectively, for some positive integer U, is also a periodic schedule, and U is referred to as the *unfold-*

ing factor of the schedule. For example, the unfolding factor of the schedule $A(2B)A(2B)(8C)$ is 2.

A related term, which will be used extensively in this book, is the *blocking factor* of a periodic schedule. The blocking factor of a periodic schedule is simply the unfolding factor of a *blocked schedule*, which is an infinite repetition of a periodic schedule in which each cycle of the schedule must complete before the next cycle is begun. The distinctions between blocked and non-blocked periodic schedules are only relevant in a parallel scheduling context, and for parallel schedules, the increased flexibility offered by a non-blocked schedule can often provide more throughput than is possible with any blocked schedule. For example, Lee [Lee86] presents a homogeneous SDF graph, shown in Figure 1.6, that can be executed at a throughput of 0.5 minimal schedule periods per time unit (assuming that each actor takes unit time to execute) with a non-blocked schedule, and Lee shows that the best throughput attainable with a blocked schedule for this graph is $U/(2U+1)$, where U is the blocking factor. Thus, for Lee's example, a blocked schedule cannot match the performance of the given unblocked schedule for any finite blocking factor.

If inline code generation is performed and no looping is applied within a period of the periodic schedule, then the total amount of code space (across all processors) required to implement a general parallel periodic schedule is roughly proportional to the unfolding factor.

Also, given a representation of a computation as a homogeneous SDF graph, there is a fundamental upper bound on the throughput. This upper bound, which was established by Reiter in [Reit68], can be computed as the minimum over all directed cycles of the number of delays in a cycle divided by the sum of

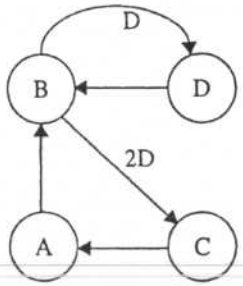


Figure 1.6. A homogenous SDF graph that does not have a rate-optimal blocked schedule for any finite blocking factor

the computation times of all actors in the cycle. A multiprocessor schedule is called *rate-optimal* if it attains this throughput bound, and the reciprocal of the rate-optimal throughput is called the *iteration period bound*.

Thus, for a given homogeneous SDF graph, it is natural to ask whether a rate-optimal schedule is attainable with a finite unfolding factor, and if so, what the minimum unfolding factor that achieves the optimum throughput is. This question was answered in the affirmative for the case where there is no restriction on the number of processors by Reiter in [Reit68]. This rate-optimal scheduling problem can be formulated as an integer linear program with only difference constraints; this class of integer programming problems is solvable in polynomial time by using the Bellman-Ford algorithm for shortest paths (and variants thereof). The unfolding factor that achieves rate-optimal throughput is given by the denominator of maximum cycle mean expressed as a rational number in reduced form (it is normally assumed that execution times of actors are integers since they are expressed in units of processor cycles). A related problem is addressed by Murthy in [Murt94b] for the more restricted class of blocked schedules. It is shown that for a given homogenous SDF graph, we can determine in a finite number of steps whether or not there is a finite blocking factor for which a rate-optimal blocked schedule exists, and when such a blocking factor exists, we can determine in a finite number of steps the minimum blocking factor for which rate-optimal blocked schedules exist. If a rate-optimal blocking factor does not exist, we can still determine those blocking factors that will give us the best throughput. That is, these problems of determining optimal blocking factors for blocked schedules are decidable.

1.8.2. Compiler Transformations

In contrast to the primary objective of minimum code size, many compilers for procedural languages apply transformations that deliberately increase the code size. One example is the inlining of subroutines, which was discussed in Section 1.6. Second, in *trace scheduling*, compile-time branch prediction is performed to estimate the most likely execution path through a program, and this path, called a *trace*, is scheduled as if it were a single basic block [Fish84]. This permits exploitation of the most abundant source of instruction-level parallelism — reordering code across basic block boundaries. For each instruction that moves across a basic block boundary, recovery code may have to be inserted just off the trace. For example, if along the trace, it is assumed that a particular conditional branch will

be taken, then each instruction migrated from after the branch to a point before the branch may have to be "undone" if the branch is not taken. The insertion of such recovery instructions increases the code size. Once the most likely trace has been selected and reorganized, the next most likely trace is selected, and the process is repeated for any desired number of traces.

In *loop unrolling*, which is analogous to the unfolding of SDF graphs, the body of a loop is replicated to cover more than one iteration, the iteration count is modified, and possibly a prologue or epilogue is generated to guarantee that the unrolled version executes the correct number of iterations of the original loop [Dong79]. As with unfolding, unrolling facilitates the exploitation of inter-iteration parallelism at the expense of a roughly linear increase in code size.

A fourth example of a code-increasing program transformation is the duplication of code to eliminate unconditional branches [Muel92]. A significant number of unconditional jumps is generated by typical compilers. For example, when generating code for an if-then-else construct, compilers often place an unconditional branch at the end of the *then* section that skips over the else section. Here, the unconditional branch can be eliminated by appending to the *then* section a duplicate copy of the code at the target of the branch. The benefits of such code replication include fewer instructions executed, better program locality and increased opportunities for common subexpression elimination [Muel92].

In the application domain that this book is concerned with — the domain of embedded real-time digital signal processing systems — the price paid for neglecting opportunities such as trace scheduling, loop unrolling, subroutine inlining or unconditional branch elimination is usually dominated by the penalty incurred when the target program does not fit within the on-chip memory limits. Given an SDF graph, we would like to first generate an efficient compact implementation, and then, if there is any remaining on-chip program memory, we can expand the code in a controlled manner to utilize it. This book focuses largely on the first part of this process — generating an efficient uniprocessor implementation with a minimal amount of code space.

1.8.3. Minimum Activation Schedules

The techniques in this book focus on compiling SDF graphs to minimize the code size and data memory size. At the Aachen University of Technology, as part of the COSSAP software synthesis environment for DSP, Ritz *et. al* have investigated the minimization of code size in conjunction with a different secondary optimization criterion: minimization of the context-switch overhead, or the average rate at which **actor activations** occur [Ritz93]. An actor activation occurs whenever two distinct actors are invoked in succession; for example, the schedule $(2(2B)(5A))(5C)$ results in five activations per schedule period. Activation overhead includes saving the contents of registers that are used by the next actor to invoke, if necessary, and loading state variables and buffer pointers into registers. In the code generation system described in [Ritz93], the context-switch overhead also includes a function call, which in turn requires saving the current value of the program counter (the return address of the function call), branching to the location of the function, retrieving the return address when the function is completed, and branching to that return address.

COSSAP gives first priority to constructing single appearance schedules and second priority to minimizing activation overhead. In other words, it attempts to construct the single appearance schedule that has the least number of activations. In contrast, the techniques developed in this book attempt to construct the single appearance schedule that minimizes the amount of memory needed for buffering tokens. When there is enough memory to accommodate the minimum activation schedules of [Ritz93], it is possible that these schedules will result in somewhat higher throughput than the schedules discussed in this paper, although the difference in context-switch overhead can often be significantly mitigated by the techniques described in [Powe92]. However, when memory constraints are severe, the techniques discussed in this book are superior.

2

Terminology and Notation

We adopt the convention of indexing vectors and matrices using functional notation rather than subscripts or superscripts. Thus, for example, $\mathbf{x}(3)$ represents the third component of the vector $\mathbf{x}$, and $M(i,j)$ represents the value at the ith row and jth column of the matrix M. We denote the transpose of the vector $\mathbf{x}$ by $\mathbf{x}^T$.

Given a finite set P of positive integers, we denote by $gcd(P)$ the greatest common divisor of P — the largest positive integer that divides all members of P, and we denote the least common multiple of the members of P by $lcm(P)$. If $gcd(P) = 1$, we say that the members of P are **coprime**. Given a finite set R of real numbers, we denote the largest and smallest numbers in R by $max(R)$ and $min(R)$, respectively. Given a fraction $f = a/b$, we define $numer(f) = a$ and $denom(f) = b$; and given a positive rational number q, $ReducedFraction(q)$ denotes that unique fraction f for which $numer(f)$ and $denom(f)$ are positive and coprime, and $(numer(f))/(denom(f)) = q$. Also, if r is a real number, we denote the largest integer that is less than or equal to r (the "floor" of r) by $\lfloor r \rfloor$, and we denote the smallest integer that is greater than or equal to r (the "ceiling" of r) by $\lceil r \rceil$. Finally, given two arbitrary sets S_1 and S_2, we define the difference of these two sets by $S_1 - S_2 \equiv \{s \in S_1 | s \notin S_2\}$. We denote the number of elements in a finite set S by $|S|$.

2.1 Graph Concepts

This section introduces the basic graph-theoretic concepts that will be applied in this monograph. For elaboration on any of these concepts, the reader is referred to [Corm90].

By a **directed multigraph**, we mean an ordered pair (V,E), where V and E are finite sets, and associated with each $e \in E$ there are two properties $src(e)$ and $snk(e)$ such that $src(e), snk(e) \in V$. Each member of V is called a **vertex** of the directed multigraph and each member of E is called an **edge**. We say that a directed multigraph is **trivial** if it contains only one vertex. If e is an edge in a directed multigraph, we say that $src(e)$ is the **source** vertex of e; $snk(e)$ is the **sink** vertex of e; e is **directed from** $src(e)$ **to** $snk(e)$; e is an **output edge** of $src(e)$; and e is an **input edge** of $snk(e)$. We represent a directed multigraph pictorially by drawing a circle for each vertex, and for each edge e, drawing a directed arc from the circle corresponding to $src(e)$ to the circle corresponding to $snk(e)$. For example, the directed multigraph depicted in Figure 2.1 consists of vertices v_1, v_2, v_3, v_4, and edges e_1, e_2, e_3, e_4, e_5, where $src(e_1) = v_1$, $snk(e_1) = v_2$, $src(e_2) = v_2$, $snk(e_2) = v_3$, $src(e_3) = v_1$, $snk(e_3) = v_3$, $src(e_4) = v_1$, $snk(e_4) = v_3$, $src(e_5) = v_4$, and $snk(e_5) = v_4$.

Given two not necessarily distinct vertices v_1 and v_2 in a directed multigraph (V,E), we say that v_1 is a **predecessor** of v_2 if there exists $e \in E$ such that $src(e) = v_1$ and $snk(e) = v_2$; we say that v_1 is a **successor** of v_2 if v_2 is a predecessor of v_1; and we say that v_1 and v_2 are **adjacent** if v_1 is a successor or predecessor of v_2; and if v_1, v_2 are distinct, then $\{v_1, v_2\}$ is called an **adjacent pair**. Two subsets $V_1, V_2 \subseteq V$ are adjacent if there exist vertices $v_1 \in V_1$ and $v_2 \in V_2$ such that v_1 and v_2 are adjacent. By a **subgraph** of a directed multigraph $G = (V, E)$, we mean the directed multigraph formed by any $V' \subseteq V$ together with the set of edges $\{e \in E | (src(e), snk(e) \in V')\}$. We denote the subgraph associated with the vertex-subset V' by $subgraph(V', G)$; if G is understood from context, we may simply write $subgraph(V')$. A **path** in (V,E) is a nonempty sequence $e_1, e_2, e_3, \ldots \in E$ such that $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, ... We say that the path $p = e_1, e_2, e_3, \ldots$ **passes through** each member of

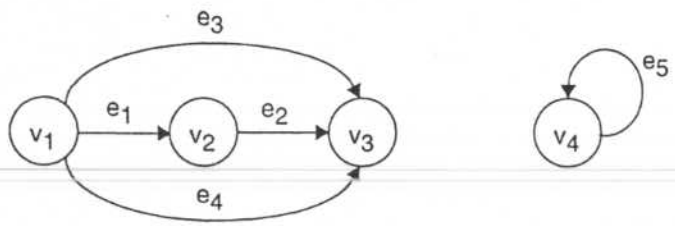


Figure 2.1. A directed multigraph.

$$Z_p = (\bigcup_i \{src(e_i)\}) \cup (\bigcup_i \{snk(e_i)\}),$$

and we refer to the SDF graph formed by Z_p together with the set of edges in p as the **associated graph** of p. Observe that the associated graph of p is not necessarily a subgraph since it does not necessarily contain all of the edges whose source and sink vertices are members of Z_p. Given a finite path $p = e_1, e_2, \ldots, e_n$, we say that p is *directed from* $src(e_1)$ *to* $snk(e_n)$. A path that is directed from some vertex to itself is called a **cycle** or a **directed cycle**, and a **fundamental cycle** is a cycle of which no proper subsequence is a cycle. A directed multigraph is **acyclic** if it contains no cycles. Finally, if e is the only edge directed from $src(e)$ to $snk(e)$, then we occasionally denote e by $src(e) \rightarrow snk(e)$; thus, in Figure 2.1, $v_1 \rightarrow v_2$ represents the edge e_1, whereas $v_1 \rightarrow v_3$ cannot represent e_3 because e_4 also has the same source and sink vertices.

If $(p_1, p_2, \ldots, p_k)$ is a finite sequence of finite paths such that $p_i = (e_{i,1}, e_{i,2}, \ldots, e_{i,n_i})$, for $1 \le i \le k$, and $snk(e_{i,n_i}) = src(e_{i+1,1})$, for $1 \le i \le (k-1)$, then we define

$$\langle (p_1, p_2, \ldots, p_k) \rangle \equiv (e_{1,1}, \ldots, e_{1,n_1}, e_{2,1}, \ldots, e_{2,n_2}, \ldots, e_{k,1}, \ldots, e_{k,n_k}).$$

Clearly, $\langle (p_1, p_2, \ldots, p_k) \rangle$ is a path from $src(e_{1,1})$ to $snk(e_{k,n_k})$. If there is a path from $v_i \in V$ to $v_j \in V$, then we say that v_i is an **ancestor** of v_j, and v_j is a **descendant** of v_i. If v_i is neither a descendant nor an ancestor of v_j, we say that v_i is **independent of** v_j. Given a directed multigraph and an vertex v_j in this graph, $ancs(v_j)$ denotes the set consisting of v_j and the set of ancestors of v_j, $desc(v_j)$ denotes the set consisting of v_j and the set of descendants of v_j, and $independent(v_j)$ denotes the set of vertices independent of v_j. Finally, an edge is **transitive** if there exists another path between the source and sink actors of the transitive edge.

A sequence of vertices $(v_1, v_2 \ldots, v_k)$ is a *chain* that joins v_1 and v_k if v_{i+1} is adjacent to v_i for $i = 1, 2, \ldots, (k-1)$. We say that a directed multigraph is

connected if for any pair of distinct members A, B of Z, there is a chain that joins A and B. Thus, the directed multigraph in Figure 2.1 is not connected, while the subgraph associated with $\{v_1, v_2, v_3\}$ is connected. Given a directed multigraph $G = (V, E)$, there is a unique partition (unique up to a reordering of the members of the partition) $V_1, V_2, \ldots, V_n$ such that for $1 \le i \le n$, $subgraph(V_i)$ is connected; and for each $e \in E$, $src(e), snk(e) \in V_j$ for some j. Thus, each V_i can be viewed as a maximal connected subset of V, and we refer to each V_i as a **connected component** of G. For example, the connected components of the directed multigraph in Figure 2.1 are $\{v_1, v_2, v_3\}$ and $\{v_4\}$. Depth-first search can be used to find the connected components of a directed multigraph in time that is linear in the number of vertices and edges.

A directed multigraph (V,E) is **strongly connected** if for each pair of distinct vertices v_1, v_2, there is a path directed from v_1 to v_2 *and* there is a path directed from v_2 to v_1. We say that a subset V' of vertices in V is strongly connected if $subgraph(V',(V,E))$ is strongly connected. A **strongly connected component** of (V,E) is a strongly connected subset $V' \subseteq V$ such that no strongly connected subset of V properly contains V'. For example, the directed multigraph in Figure 2.2 has three strongly connected components - $\{v_1, v_2\}$, $\{v_3, v_4, v_5\}$, and $\{v_6\}$. The set $\{v_4, v_5\}$ is strongly connected but is not a strongly connected component. The strongly connected components of a directed multigraph can be determined in linear time by an algorithm developed by Tarjan [Tarj72].

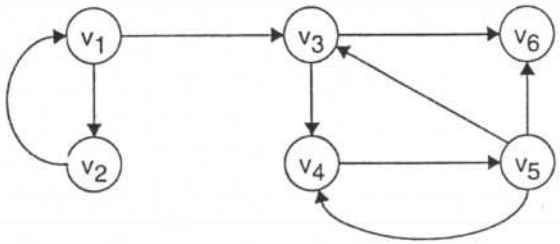


Figure 2.2. A directed multigraph that has three strongly connected components — $\{v_1, v_2\}$, $\{v_3, v_4, v_5\}$, and $\{v_6\}$.

Given a directed multigraph (V, E), a vertex v of (V, E) is a **root vertex** of (V, E) if there is no edge e in (V, E) such that $snk(e) = v$. A **root strongly connected component** of (V, E) is a strongly connected component V' of (V, E) such that $\{e \in E | (src(e) \notin V' \text{ and } snk(e) \in V')\} = \varnothing$. For example, in Figure 2.2 there are no root vertices, and there is one root strongly connected component — $\{v_1, v_2\}$. Finally, if V' is a connected component of (V,E), then $subgraph(V')$ is called a **connected component subgraph** of (V,E); similarly, if V' is a strongly connected component of (V,E), then $subgraph(V')$ is a **strongly connected component subgraph** of (V,E).

A **topological sort** of an acyclic directed multigraph (V,E) is an ordering $v_1, v_2, \ldots, v_{|V|}$ of the members of V such that for each $e \in E$, $((src(e) = v_i) \text{ and } (snk(e) = v_j) \Rightarrow (i < j))$; that is, the source vertex of each edge occurs earlier in the ordering than the sink vertex. Thus, in Figure 2.2, $subgraph(\{v_1, v_3, v_4, v_6\})$ has two distinct topological sorts — (v_1, v_3, v_4, v_6) and (v_1, v_3, v_6, v_4). An acyclic directed multigraph is said to be **well-ordered** if it has only one topological sort, and we say that an n-vertex well-ordered directed multigraph is **chain-structured** if it has $(n-1)$ edges. Thus, for a chain-structured directed multigraph, there are orderings $v_1, v_2, \ldots, v_n$, and $e_1, e_2, \ldots, e_{n-1}$ of the vertices and edges, respectively, such that each e_i is directed from v_i to v_{i+1}. For example, in Figure 2.2, $subgraph(\{v_1, v_3, v_6\})$ is chain-structured, while $subgraph(\{v_3, v_4, v_6\})$ is neither chain-structured nor well-ordered; and in Figure 2.1, $subgraph(\{v_1, v_2, v_3\})$ is well-ordered but not chain-structured.

In the remainder of this book, by a "graph" or a "directed graph", we mean a directed multigraph, unless otherwise stated.

2.2 Computational Complexity

When discussing the complexity of algorithms, we will use the standard O, Ω, and, Θ notation. A function $f(x)$ is $O(g(x))$ if for sufficiently large x, $f(x)$ is bounded above by a positive real multiple of $g(x)$. Similarly, $f(x)$ is $\Omega(g(x))$ if $f(x)$ is bounded below by a positive real multiple of $g(x)$ for sufficiently large x. Finally, $f(x)$ is $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$.

In the following paragraphs, we give an informal discussion of the theory of NP-completeness for the benefit of those readers completely unfamiliar with the subject. The theory can be studied in rigorous detail in any number of textbooks such as [Corm90] and [Gare79].

Let x represent the size of an input instance to some algorithm A. Suppose that the running time of this algorithm, $f(x)$, is represented as $\Theta(g(x))$. If the function $g(x)$ is a polynomial function of x, then the algorithm is said to run in **polynomial time**. If $g(x)$ is an exponential function of x, then the algorithm is said to run in **exponential time**. Given a problem instance, the solution returned by the algorithm is called the **certificate**.

The size of an input instance is defined as the size of the data-structure needed to represent the input instance using *some reasonable form of encoding*. For example, if the input instance is just a graph (a set of vertices and a set of edges), then the size of any data-structure needed to represent this graph will be proportional to the number of vertices and the number of edges. If the input instance happens to have numbers as well, then the last part of the definition becomes crucial: any number N appearing in the input instance will be represented by $\log(N)$ bits; this is a reasonable form of encoding for a number. In particular, using N bits to represent N (in unary notation for example) is **not** a reasonable form of encoding. Hence, for input instances involving numbers, the size of the input instance will be proportional to the *logarithm* of those numbers and not the numbers directly.

Polynomial time algorithms are desirable over exponential time algorithms for obvious reasons: exponential time algorithms will take impractically long times to run on input instances of even moderate sizes. For example, consider two algorithms, one of which runs in time (expressed in microseconds) $f(n) = n^5$ and another in time $f(n) = 2^n$. If $n = 20$, then the first algorithm will take 3.2 seconds while the second algorithm will take only 1 second. However, if $n = 60$, then the first algorithm will take 13.0 minutes while the second will take 366 centuries.

Not all combinatorial problems have polynomial time algorithms. The class of problems that can be solved in polynomial time is called the class **P**. A more general class of problems, the class **NP**, are those that have the following property: a certificate (solution) to the problem can be *verified* in polynomial time. This simply means that regardless of how long the computation of the certificate takes, once the certificate is there, we can check whether or not it is in fact a solution to the problem in polynomial time. For example, if the combinatorial problem is to sort N integers, then a certificate for this problem would be the N integers in some particular order. In order for us to check whether this certificate is in fact legitimate, we would have to check whether a) each integer in the solution appears as many times as it does in the input (that is, the set of integers returned as the solution is a permutation of the set of input integers), and b) the set of integers is in increasing order. Both of these steps can be performed in polynomial time; hence, the sorting problem belongs to the class **NP**.

Notice that any problem in **P** automatically belongs in **NP** since the validity of any certificate is established in polynomial time by actually computing it in polynomial time. Hence, **P** $\subseteq$ **NP**. One of the biggest unsolved problems in computer science is whether these classes are equal, or whether the containment is strict. In order to prove that a problem is in **P**, we have to exhibit a polynomial time algorithm for solving it. There are a great many combinatorial problems that we can easily show to be in **NP** but cannot show to be in **P** because we are unable to come up with algorithms that solve the problem in polynomial time. All known algorithms for these problems take exponential time. For reasons too technical to go into here, if **NP** $\neq$ **P**, then there is a subset of **NP** disjoint from **P** called the set of **NP-complete problems**. This class of problems has the property that if any problem in this class is shown to have a polynomial time algorithm, then **NP** = **P**. Since this class is very rich (has many problems belonging to it), and many researchers have attempted to solve them in polynomial time but have been unable to do so, showing that a problem is NP-complete (that is, it belongs to the set of NP-complete problems) is tantamount to showing that it is highly unlikely that a polynomial time algorithm exists for the problem. Hence, proving a problem to be NP-complete forces us to abandon our search for algorithms that will give the exact solution in polynomial time. Instead, we resort to **heuristics**: algorithms that always run in polynomial time, but may or may not give us the exact solution. Typically, we are interested in optimization problems where we want the solution that maximizes or minimizes some cost function; a good heuristic will return, in general, a solution that is quite close to the optimal one.

A problem B is said to be **polynomial-time reducible** to problem A if there exists a polynomial time algorithm that transforms an arbitrary instance of problem B to an instance of problem A such that a solution for A implies a solution for B and vice versa. Formally, a problem A is defined to be NP-complete if two conditions are satisfied:

a) $A \in$ **NP**, and

b) B is polynomial-time reducible to A for every B in **NP**.

Fortunately, we do not have to use this definition to show that a problem is NP-complete (trying to reduce *every* problem in **NP** would be highly tedious at best!). Cook's theorem (see [Gare79]) proves that a problem called CIRCUIT-SAT is NP-complete. This allows us to use the following alternative definition of NP-completeness:

a) $A \in$ **NP**, and

b) B is polynomial time reducible to A, for some B that is NP-complete.

The reason why this definition is equivalent to the earlier definition is the following. Notice that since B is NP-complete, every problem in **NP** reduces to B. Since B reduces to A, every problem in NP reduces to A (polynomial-time reductions are transitive). Cook's theorem guarantees that there is at least one NP-complete problem; from this problem, a whole library of problems has been built up. This allows us to choose a problem that has already been proven to be NP-complete and reduce that to our problem.

So if we show that A is NP-complete, then it is unlikely that A has a polynomial-time solution because if it were to have one, then every problem in **NP** would have one. As mentioned earlier, this is believed to be unlikely.

Note also that if A is NP-complete, then $A \in$ **NP**, meaning that A reduces to B implicitly, since B is NP-complete. Hence, the reduction only needs to be demonstrated one way.

3

Synchronous dataflow

Recall that in a dataflow graph, the data values on the edges are referred to as **tokens**. Formally, an SDF graph is a directed multigraph in which each edge α has three properties in addition to $src(\alpha)$ and $snk(\alpha)$:

- $del(\alpha)$, a nonnegative integer that gives the number of initial tokens associated with α. The number of initial tokens $del(\alpha)$ is also called the **delay** of an edge because it can induce an offset in the execution of the sink actor in relation to the source actor.
- $prd(\alpha)$, a positive integer that indicates the number of tokens produced onto the channel corresponding to α by each execution of the computation corresponding to $src(\alpha)$.
- $cns(\alpha)$, a positive integer that represents the number of tokens consumed from α by each execution of $snk(\alpha)$.

We refer to a vertex of an SDF graph as an **actor**, and given an SDF graph G, we represent the set of actors and the set of edges in G by $actors(G)$ and $edges(G)$, respectively. If for each $\alpha \in edges(G)$, $prd(\alpha) = cns(\alpha) = 1$, then we say that G is a **homogeneous** SDF (HSDF) graph.

Conceptually, each edge in G, corresponds to a FIFO (first-in, first-out) queue that buffers the tokens that pass through the edge. The FIFO queue associated with an edge is called a **buffer** for that edge, and the process of maintaining the queue of tokens on a buffer is referred to as **buffering**. Each buffer contains an initial number of tokens equal to the delay on the associated edge. A **firing** of

an actor in G corresponds to removing $cns(\alpha)$ tokens from the head of the buffer for each input edge α, and appending $prd(\beta)$ tokens to the buffer for each output edge β. Thus, a firing is only possible if for each input edge α, there are at least $cns(\alpha)$ tokens on the corresponding buffer. After a sequence of 0 or more firings, we say that an actor is **fireable** if there are enough tokens on each input buffer to fire the actor. A **schedule** for G is a sequence $S = f_1 f_2 f_3 \ldots$, which can be finite or infinite, of actors in G. Each term f_i of this sequence is called an **invocation** of the corresponding actor in the schedule; and for each actor N, we denote the jth invocation of N in the schedule by N_j, and we call j the **invocation number** of N_j. The schedule that consists of no invocations — the empty sequence — is called the **null schedule**. For each i, f_i is said to be an **admissible firing** if it is fireable immediately after $f_1, \ldots, f_{i-1}$ have fired in succession. The schedule $S = f_1 f_2 f_3 \ldots$ is an **admissible schedule** for G if f_i is an admissible firing for each i. The process of successively firing the invocations in an admissible schedule is called **executing** the schedule, and if a schedule is executed repeatedly, each repetition of the schedule is called a **schedule period** of the execution.

If e is an SDF edge, then the **delayless version** of e is an edge e' such that $e' = e$ if $del(e) = 0$, and if $del(e) \neq 0$, then e' is the edge defined by $src(e') = src(e)$, $snk(e') = snk(e)$, $cns(e') = cns(e)$, $prd(e') = prd(e)$, and $del(e') = 0$. If $G = (V, E)$ is an SDF graph, then G is **delayless** if $del(e) = 0$ for all $e \in E$, and the **delayless version** of G is the SDF graph defined by (V, E'), where $E' = \{\text{the delayless version of } e \mid e \in E\}$. In words, the delayless version of G is the graph that results from setting the delays on all edges to zero.

Consider the simple SDF graph in Figure 3.1. Each edge is annotated with the number of tokens produced by its source actor and the number of tokens consumed by its sink — for example, actor A produces three tokens per firing on its output edge and B consumes two tokens from its input edge. The "2D" next to the edge directed from A to B indicates that this edge has a delay of 2. Now consider the schedule $BACBA$ for this example. As we fire the invocations in the schedule, we can represent the **state** of the system — the number of tokens

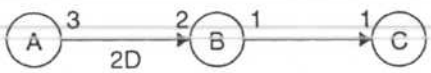


Figure 3.1. A simple SDF graph.

queued on the buffers — with an ordered pair whose first and second members are, respectively, the number of tokens on the edge $A \rightarrow B$ and the number of tokens on the edge $B \rightarrow C$. Then, since there is a delay of 2 on the left side edge, the initial state of the system is $(2,0)$. Thus, at the start, B is fireable, and as it fires, two tokens are removed from the left edge and one token is appended to the right edge, so the state becomes $(0,1)$. Since A has no input edges, it can be fired at any time. Hence the second invocation of the schedule is an admissible firing and its firing leads to the state $(3,1)$. It is easily verified that the remaining three invocations in the schedule are admissible firings, and the sequence of buffer states that results from these remaining firings is $(3,0), (1,1), (4,1)$. Thus, $BACBA$ is an admissible schedule for the SDF graph in Figure 3.1. In contrast, the slightly different schedule $BACBB$ is not admissible, since only one token resides on the input edge of B prior to the third invocation of B, so B is not fireable.

If $S = A_1 A_2 A_3 \ldots$ is not an admissible schedule, then some A_i is not fireable immediately after its antecedents have fired. Thus, there is at least one edge α such that (1) $snk(\alpha) = A_i$ and (2) the buffer associated with $snk(\alpha)$ contains less than $cns(\alpha)$ tokens just prior to the ith firing in S. For each such α, we say that S **terminates on α at invocation** A_i. Clearly then, a schedule is admissible if and only if it does not terminate on any edge.

Given a schedule S and an actor A, we define $inv(A, S)$ to be the number of times that S invokes A. For example, $inv(A, BACBA) = 2$.

We say that a finite schedule S is a **periodic schedule** if it invokes each actor at least once and produces no net change in the system state (i.e, the number of tokens on each edge is the same after S has executed as before) — for each edge α, $(inv(src(\alpha), S)) \times prd(\alpha) = (inv(snk(\alpha), S)) \times cns(\alpha)$. For example for the SDF graph in Figure 3.1, we saw that if the initial state is $(2,0)$, the state after executing the schedule $BACBA$ is $(4,1)$. Thus this schedule produces a net change of $+2$ tokens on edge $A \rightarrow B$ and $+1$ token on $B \rightarrow C$, so this schedule is not periodic.

Suppose that $\mathbf{b}$ is a vector of positive integers indexed by the actors in a connected SDF graph G. Non-connected SDF graphs can be analyzed by examining each connected component separately; we will elaborate on this in Section 4.3. By definition, a schedule that invokes each actor A $\mathbf{b}(A)$ times is a periodic schedule if and only if for each edge α in G,

$$\mathbf{b}(src(\alpha)) \times prd(\alpha) = \mathbf{b}(snk(\alpha)) \times cns(\alpha). \tag{3.1}$$

This system of equations in the set of variables $\{\mathbf{b}(A) | (A \in actors(G))\}$ — consisting of one equation for each edge in G — is known as the system of **balance equations** for G. Clearly, a periodic schedule exists for G if and only if the balance equations have a solution whose components are all positive integers[1]. The balance equations can be expressed more compactly in matrix-vector form as

$$\Gamma \mathbf{b} = \mathbf{0}, \tag{3.2}$$

where Γ, called the **topology matrix** of G, is a matrix whose rows are indexed by the edges in G and whose columns are indexed by the actors in G, and whose entries are defined by

$$\Gamma(\alpha, A) = \begin{cases} prd(\alpha), & \text{if } A = src(\alpha) \\ -cns(\alpha), & \text{if } A = snk(\alpha) \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

This formulation assumes that G does not contain any **self-loops**, edges whose source and sink vertices are identical, such as edge e_5 in Figure 2.1. In such a case, there is a vertex α such that $src(\alpha) = snk(\alpha)$, and therefore equation 3.3 is self contradictory. In an SDF graph, a self-loop edge α precludes the existence a periodic schedule if $prd(\alpha) \neq cns(\alpha)$; otherwise it has no effect on the existence of a periodic schedule, and thus it can be ignored in this analysis.

A periodic schedule exists if and only if equation 3.2 has a solution $\mathbf{b}$ where each element of the vector $\mathbf{b}$ is a positive integer. From equation 3.2, it is evident that this requires that the topology matrix Γ not have full rank. We will now show that for a connected graph G, the rank of Γ is always either n or $n-1$, where n is the number of actors in G [Lee87]. Thus, for a periodic schedule to exist, it is necessary for Γ to have rank $n-1$.

First, observe that there are many possible topology matrices describing a given graph, depending on how we number the actors and edges to index the rows and columns. However, all such matrices have the same rank. Any topology matrix can be constructed from any other for the same graph by row and column exchanges, and these do not affect the rank.

[1] Recall that in our definition of *periodic schedule*, we do not require admissibility — a periodic schedule need not be admissible.

Next, consider a tree-structured graph. This is a connected graph with no cycles, ignoring the direction of the edges.

Lemma 3.1: A connected, tree-structured graph with n vertices has exactly $n-1$ edges.

Proof: The proof is by induction. A 2-vertex graph is tree-structured and connected iff it has one edge connecting the 2 vertices. Now consider an n-vertex tree-structured, connected graph. By the induction hypothesis, this graph has $n-1$ edges. Suppose we add another actor t to this graph. We have to add at least one edge to keep the graph connected. Adding one edge from the new vertex to any other vertex in the graph will not introduce a cycle because the n-vertex graph does not have cycles, and the new vertex has only one edge incident to it. Suppose we add two edges from the new vertex to two other vertices u, v in the n-vertex graph. Since the n-vertex graph is connected, there is a path from u to v. The newly formed path $v \leftrightarrow t \leftrightarrow u$ thus creates a cycle and destroys the tree-structure. This shows that for each actor added to a tree-structured, connected graph, we have to add one edge, proving the lemma. ■

Lemma 3.2: A topology matrix Γ for a tree-structured SDF graph G with n actors has rank $n-1$.

Proof: We prove this by induction. The lemma is clearly true for a two-actor graph. Assume the lemma is true for some n-actor tree-structured graph with topology matrix Γ_n. Adding one more actor and one more edge connecting the new actor to the graph will yield an $n+1$ actor tree-structured graph. A topology matrix for the new graph can be constructed from the old by adding one column for the new actor and one row for the new edge, as follows:

$$\Gamma_{n+1} = \left[\begin{array}{c|c} \Gamma_n & \mathbf{0} \\ \hline \multicolumn{2}{c}{\rho^T} \end{array} \right], \tag{3.4}$$

where $\mathbf{0}$ is a column vector full of zeros, and ρ^T is a row vector corresponding to the new edge. The last entry in ρ^T must be nonzero since it indicates the number of tokens consumed or produced by the new actor on the new edge. Hence, the last row is linearly independent of the other rows, so $\text{rank}(\Gamma_{n+1}) = \text{rank}(\Gamma_n) + 1$. Since any tree-structured graph can be constructed by starting with a two-actor graph and adding one more actor and edge repeatedly, $\text{rank}(\Gamma_n) = n-1$ for any n. ■

Lemma 3.3: A topology matrix Γ for a connected SDF graph G with n actors either has rank $n-1$ or n.

Proof: First, it is obvious that the topology matrix cannot have rank greater than n, because it has only n columns. We now show that it cannot have rank less than $n-1$. Consider any spanning tree of the graph (a spanning tree is a tree that includes every actor in the graph). This spanning tree is a subgraph. It has all the actors of the original graph, but only a subset of the edges. Let Γ_T be the topology matrix for this subgraph. From lemma 3.2, $\text{rank}(\Gamma_T) = n-1$. Adding edges to the subgraph will simply add rows to the topology matrix. Adding rows to a matrix can increase the rank, if the rows are linearly independent of prior rows, but it cannot decrease it. Therefore, the rank of Γ cannot be less than $n-1$. ■

Since for any connected SDF graph the rank of Γ must be n or $n-1$, and for equation 3.2 to have a solution we need for Γ to not have rank n (full rank), we conclude that for a periodic schedule to exist, it is necessary for Γ to have rank $n-1$. We next show that this is sufficient by showing that whenever the rank is $n-1$, a positive-integer vector $\mathbf{b}$ exists that satisfies the balance equations.

Lemma 3.4: Consider a connected SDF graph G with topology matrix Γ. Let $\mathbf{b}$ be any vector such that $\Gamma\mathbf{b} = \mathbf{0}$. Denote any connected path of length L through the graph by the set of actors $A = \{A_1, ..., A_L\}$, where A_i is connected to A_{i+1} for $1 \le i < L$. Then in the set $\{\mathbf{b}(a), a \in A\}$, either all members are zero, all are positive, or all are negative. Moreover, if any member of the set is rational, then all members are rational.

Proof: Again, we use induction. Consider a connected path with two actors, $A = \{A_1, A_2\}$, and let α denote the edge that connects them. Assume without loss of generality that the edge is directed from A_1 to A_2. From equation 3.1,

$$\mathbf{b}(A_1)prd(\alpha) = \mathbf{b}(A_2)cns(\alpha). \tag{3.5}$$

Since both $prd(\alpha)$ and $cns(\alpha)$ are positive integers, the lemma follows immediately. Moreover, given that the lemma is true for some path of length L, proving it is true for a path of length $L+1$ that extends the original path is straightforward, using the same argument. ■

In a connected graph, there is a path from any actor to any other. Thus, the following corollary follows immediately from lemma 3.4:

Corollary 3.1: For any $\mathbf{b}$ such that $\Gamma\mathbf{b} = \mathbf{0}$, all entries are either zero, positive, or negative. Moreover, if any one is rational, then all are rational.

We can now show that if Γ has rank $n-1$, then there is a unique minimum vector, denoted $\mathbf{q}_G$, that has all positive integer entries and satisfies $\Gamma\mathbf{q}_G = \mathbf{0}$. This vector is called the **repetitions vector** of G, and we denote it simply by $\mathbf{q}$ if G is understood from context.

First, notice that if Γ has rank $n-1$, we have the following facts from linear algebra:

Fact 3.1: There exists a vector $\mathbf{b} \neq \mathbf{0}$ such that $\Gamma\mathbf{b} = \mathbf{0}$.

Fact 3.2: If $\Gamma\mathbf{b}_1 = \mathbf{0}$ then $\Gamma(K\mathbf{b}_1) = \mathbf{0}$ for any constant K.

Fact 3.3: If $\Gamma\mathbf{b}_1 = \mathbf{0}$ and $\Gamma\mathbf{b}_2 = \mathbf{0}$, then there exists a scalar constant K such that $\mathbf{b}_1 = K\mathbf{b}_2$.

Together, these facts tell us that the null space of Γ (the set of all vectors $\mathbf{b}$ such that $\Gamma\mathbf{b} = \mathbf{0}$) has dimension one. Every vector in the null space is a scalar multiple of every other nonzero vector in the null space.

To prove the existence of a repetitions vector, we use fact 3.1 to assert that there is a solution $\mathbf{b} \neq \mathbf{0}$. From corollary 3.1, all entries in this vector are non-zero and have the same sign. In case they are irrational, let b_1 denote the first entry, and define

$$\mathbf{b}' = \frac{1}{b_1}\mathbf{b} \tag{3.6}$$

From fact 3.2, $\Gamma\mathbf{b}' = \mathbf{0}$. Since the first entry in $\mathbf{b}'$ is rational, from corollary 3.1, all entries are rational. Express each of these entries as a reduced fraction (where the numerator and denominator are relatively prime), and let η be the least common multiple of all the denominators (this can be found using Euclid's algorithm [Corm90]). Then

$$\mathbf{q} = \eta\mathbf{b}' \tag{3.7}$$

is the unique, smallest positive integer solution to equation 3.2. Uniqueness follows from fact 3.3. The integers in $\mathbf{q}$ have no common factors, and thus are said to be *coprime*.

We have proven the following theorem.

Theorem 3.1: A connected SDF graph with n actors has a periodic schedule if and only if its topology matrix Γ has rank $n-1$. Moreover, if its topology matrix Γ has rank $n-1$, then there exists a unique smallest integer solution $\mathbf{q}$ to the balance equations $\Gamma\mathbf{q} = \mathbf{0}$. Moreover, the entries in the vector $\mathbf{q}$ are coprime.

Notice that the graph may not have an *admissible* schedule. This will depend on the number of delays on edges in a directed cycle. The topology matrix, and hence the existence of a periodic schedule, does not depend on the delays in an SDF graph. Facts 3.4-3.6 summarize some key properties that follow immediately from theorem 3.1.

Fact 3.4: A positive-integer vector $\mathbf{q}$ is the repetitions vector of a connected SDF graph if and only if its components are coprime and it satisfies the balance equations, $\Gamma\mathbf{q} = \mathbf{0}$.

Fact 3.5: Any positive-integer vector that satisfies the balance equations is a positive-integer multiple of the repetitions vector.

Fact 3.6: A schedule S for a connected SDF graph G is periodic if and only if the repetitions vector $\mathbf{q}_G$ exists and there exists a positive integer J_0 such that S invokes each actor A exactly $J_0\mathbf{q}_G(A)$ times.

The positive integer J_0 in Fact 3.6 is called the **blocking factor** of the associated schedule. If S is a periodic schedule, we denote the blocking factor of S by $J(S)$, and if $J(S) = 1$ we say that S is a **minimal** periodic schedule.

An example of a connected SDF graph that does not have a periodic schedule is shown in Figure 3.2(a). The topology matrix for this SDF graph is

$$\Gamma = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}, \tag{3.8}$$

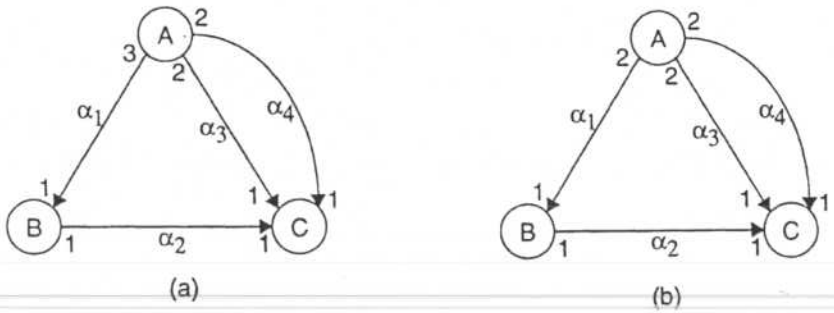


Figure 3.2. (a) An SDF graph that does not have a periodic schedule.
(b) A slightly modified version that has a periodic schedule.

where each α_i corresponds to the ith row and the ith vertex (in alphabetical order) corresponds to the ith column. Observe that the bottom two rows of Γ are identical, and the top three rows form a square matrix whose determinant is non-zero. Thus, the matrix contains three linearly independent rows, so it has full rank, and there is no nontrivial solution to 3.2.

To understand what is "defective" about this graph, observe that for each firing of A, three firings of B are required to return edge α_1 to its initial state of having no tokens queued in its buffer, and then three firings of C are required to return α_2 to its initial state. However, since α_3 is an input edge of C and A produces only two tokens per firing on α_3, only two firings of C are possible for each firing of A. Thus, any infinite admissible sequence of firings for this graph will produce an unbounded token accumulation on α_1, α_2, or both.

If we change $prd(\alpha_1)$ to 2, the resulting SDF graph, shown in Figure 3.2(b), has a periodic schedule. The topology matrix of this new SDF graph is

$$\Gamma' = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}. \tag{3.9}$$

It is easily verified that the first two rows of Γ' are linearly independent, and each of the third and fourth rows is the sum of the first two rows. Thus, the rank of Γ' is 2, one less than the number of actors, so positive-integer solutions to (3.2) exist, and thus the repetitions vector exists. The repetitions vector for Figure 3.2(b) is given by

$$\mathbf{q}(A, B, C) = (1, 2, 2)^T. \tag{3.10}$$

From (3.10), we see that $ABCBC$ and $ABBCC$ are minimal periodic schedules, and $ABABBCBCCC$ is a periodic schedule having blocking factor 2. All three of these schedules are admissible.

This book is primarily concerned with schedules that are both periodic and admissible, and we refer to such schedules as **valid** schedules. An SDF graph is **consistent** if and only if it has a valid schedule, and we say that an SDF graph is **sample rate consistent** if it has a periodic schedule. Thus, for SDF graphs, consistency implies sample rate consistency, but the converse is not true: a sample rate consistent SDF graph that is deadlocked is not consistent.

Notice that the graph may not have an *admissible* schedule. This will depend on the number of delays on edges in a directed cycle. The topology matrix, and hence the existence of a periodic schedule, does not depend on the delays in an SDF graph. Facts 3.4-3.6 summarize some key properties that follow immediately from theorem 3.1.

Fact 3.4: A positive-integer vector $\mathbf{q}$ is the repetitions vector of a connected SDF graph if and only if its components are coprime and it satisfies the balance equations, $\Gamma\mathbf{q} = \mathbf{0}$.

Fact 3.5: Any positive-integer vector that satisfies the balance equations is a positive-integer multiple of the repetitions vector.

Fact 3.6: A schedule S for a connected SDF graph G is periodic if and only if the repetitions vector $\mathbf{q}_G$ exists and there exists a positive integer J_0 such that S invokes each actor A exactly $J_0\mathbf{q}_G(A)$ times.

The positive integer J_0 in Fact 3.6 is called the **blocking factor** of the associated schedule. If S is a periodic schedule, we denote the blocking factor of S by $J(S)$, and if $J(S) = 1$ we say that S is a **minimal** periodic schedule.

An example of a connected SDF graph that does not have a periodic schedule is shown in Figure 3.2(a). The topology matrix for this SDF graph is

$$\Gamma = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}, \tag{3.8}$$

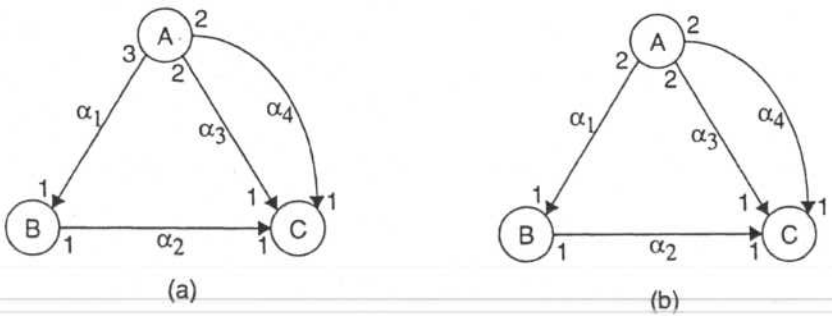


Figure 3.2. (a) An SDF graph that does not have a periodic schedule.
(b) A slightly modified version that has a periodic schedule.

where each α_i corresponds to the ith row and the ith vertex (in alphabetical order) corresponds to the ith column. Observe that the bottom two rows of Γ are identical, and the top three rows form a square matrix whose determinant is non-zero. Thus, the matrix contains three linearly independent rows, so it has full rank, and there is no nontrivial solution to 3.2.

To understand what is "defective" about this graph, observe that for each firing of A, three firings of B are required to return edge α_1 to its initial state of having no tokens queued in its buffer, and then three firings of C are required to return α_2 to its initial state. However, since α_3 is an input edge of C and A produces only two tokens per firing on α_3, only two firings of C are possible for each firing of A. Thus, any infinite admissible sequence of firings for this graph will produce an unbounded token accumulation on α_1, α_2, or both.

If we change $prd(\alpha_1)$ to 2, the resulting SDF graph, shown in Figure 3.2(b), has a periodic schedule. The topology matrix of this new SDF graph is

$$\Gamma' = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{bmatrix}. \tag{3.9}$$

It is easily verified that the first two rows of Γ' are linearly independent, and each of the third and fourth rows is the sum of the first two rows. Thus, the rank of Γ' is 2, one less than the number of actors, so positive-integer solutions to (3.2) exist, and thus the repetitions vector exists. The repetitions vector for Figure 3.2(b) is given by

$$\mathbf{q}(A, B, C) = (1, 2, 2)^T. \tag{3.10}$$

From (3.10), we see that $ABCBC$ and $ABBCC$ are minimal periodic schedules, and $ABABBCBCCC$ is a periodic schedule having blocking factor 2. All three of these schedules are admissible.

This book is primarily concerned with schedules that are both periodic and admissible, and we refer to such schedules as **valid** schedules. An SDF graph is **consistent** if and only if it has a valid schedule, and we say that an SDF graph is **sample rate consistent** if it has a periodic schedule. Thus, for SDF graphs, consistency implies sample rate consistency, but the converse is not true: a sample rate consistent SDF graph that is deadlocked is not consistent.

Clearly, an SDF graph is consistent if and only if each connected component subgraph is consistent, and a necessary condition for a connected SDF graph to be consistent is that the topology matrix does not have full rank. However, for an admissible periodic schedule to exist, an SDF graph must also have a sufficient amount of delay in each fundamental cycle. For example, consider the SDF graph in Figure 3.3. The repetitions vector for this graph is given by

$$\mathbf{q}(A, B, C) = (3, 3, 2)^T, \tag{3.11}$$

and thus periodic schedules exists. However, one can easily verify that there are only five possible non-null admissible schedules for this SDF graph — B, BA, BAC, $BACB$, and $BACBA$. Since none of these five schedules contains enough invocations for a periodic schedule, we see that a valid schedule does not exist. If we increase delay on the output edge of A from one to two, valid schedules, such as the periodic schedule $BACBACBA$, exist.

Associated with any connected, consistent SDF graph G, given a positive integer blocking factor J, there is a unique directed graph, called an **acyclic precedence graph (APG)**, that specifies the precedence relationships between actor invocations throughout J successive minimal schedule periods for G [Lee87]. Each vertex of the APG corresponds to an actor invocation. There is an edge directed from the vertex corresponding to invocation A_i to the vertex corresponding to B_j if and only if at least one token produced by A_i is consumed by B_j. As a simple example, Figure 3.4 below shows the APG for Figure 3.1 and blocking factor 1. See [Sih91] for an efficient algorithm that systematically constructs the APG.

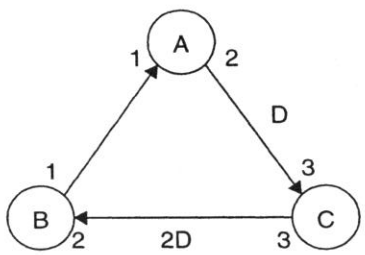


Figure 3.3. An SDF graph that has a repetitions vector but does not have an admis-

We say that two SDF graphs G_1 and G_2 are **isomorphic** if there exist bijective mappings

$$f_1 : actors(G_1) \rightarrow actors(G_2) \text{ and } f_2 : edges(G_1) \rightarrow edges(G_2)$$

such that for each $\alpha \in edges(G_1)$,

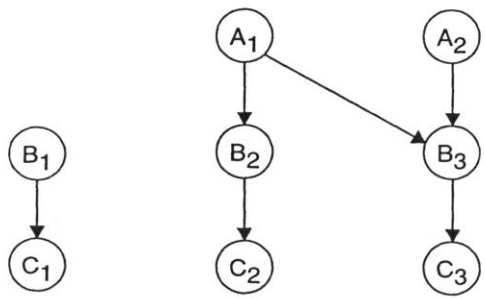
$$src(f_2(\alpha)) = f_1(src(\alpha)),\ snk(f_2(\alpha)) = f_1(snk(\alpha)),$$
$$del(f_2(\alpha)) = del(\alpha),\ prd(f_2(\alpha)) = prd(\alpha), \text{ and}$$
$$cns(f_2(\alpha)) = cns(\alpha).$$

Intuitively, two SDF graphs are isomorphic if they differ only by a relabeling of the actors and/or edges. For example, $subgraph(\{B, C\})$ in Figure 3.3 and $subgraph(\{A, B\})$ in Figure 3.1 are isomorphic.

Finally, given a sample rate consistent, connected SDF graph G and an edge α in G, we denote the total number of tokens consumed by $snk(\alpha)$ in a minimal schedule period by $TNSE(\alpha, G)$; that is

$$TNSE(\alpha, G) = \mathbf{q}_G(snk(\alpha)) \times cns(\alpha).$$

Since in a periodic schedule, the number of tokens produced on an edge equals the number of tokens consumed, we also have that $TNSE(\alpha, G) = \mathbf{q}_G(src(\alpha)) \times prd(\alpha)$. If G is understood from context, then we may suppress the second argument and write $TNSE(\alpha)$ in place of $TNSE(\alpha, G)$.



```
procedure ComputeRepetitions(G)
        for each A ∈ actors(G), initialize reps(A) to zero
        select an actor A' ∈ actors(G)
        SetReps (A', 1)
        compute x = lcm({denom(reps(A))|A ∈ actors(G)})
        for each A ∈ actors(G), reps(A) = x × reps(A)
        for each edge α ∈ edges(G)
            if (reps(src(α)) × prd(α)) ≠ (reps(snk(α)) × cns(α))
                error: inconsistent graph
                exit
            endif
        endfor
        endfor
procedure SetReps (A, n)
        reps(A) = n
        for each output edge α of A
            if reps(snk(α)) = 0
                SetReps( snk(α),
                ReducedFraction((nprd(α))/cns(α)))
            endif
        endfor
        for each input edge α of A
            if reps(src(α)) = 0
                SetReps(src(α),
                ReducedFraction((ncns(α))/prd(α)))
            endif
        endfor
```

Figure 3.5. Algorithm for computing the repetitions vector

3.1 Computing the Repetitions Vector

The repetitions vector can be computed efficiently by applying depth-first search. An algorithm based on the one that is used in the Gabriel [Lee89] and Ptolemy [Buck94] systems is described by the pseudocode segment in Figure 3.5. In this algorithm, we maintain an array of fractions called *reps*. At the end of the

algorithm, $numer(reps(A)) = \mathbf{q}_G(A)$, for each actor A in the input SDF graph G, if G has a repetitions vector.

Assuming the production and consumption parameters on the edges are bounded — so that computing the least common multiple of two numbers is an elementary operation — this algorithm has time complexity that is linear in the number of actors and edges in the input SDF graph (that is, its running time is given by $\Theta(|edges(G)| + |actors(G)|)$.

3.2 Constructing a Valid Schedule

If a connected SDF graph is consistent and the repetitions vector is computed, a valid schedule can be constructed. In [Lee87], Lee and Messerschmitt define a class of scheduling algorithms, called *class-S algorithms*, that construct valid schedules given a positive integer multiple of the repetitions vector $\mathbf{r}$ ($\mathbf{r} = k\mathbf{q}$ for some positive integer k). A class-S algorithm maintains the state of the system as a vector $\mathbf{b}$ that is indexed by the edges in the input SDF graph. A class-S algorithm is any algorithm that repeatedly schedules fireable actors, updating $\mathbf{b}$ as each actor is fired, until either no actor is fireable or until all actors have been scheduled exactly the number of times specified by the corresponding component of $\mathbf{r}$. Thus, once an actor A has been scheduled $\mathbf{r}(A)$ times, a class-S algorithm does not schedule A again. Lee and Messerschmitt show that a class-S algorithm constructs a valid schedule if and only if the SDF graph in question is consistent [Lee87].

One specific class-S scheduling algorithm is given by procedure *ConstructValidSchedule* in Figure 3.6. It is easily verified that if we assume that the number of input and output edges for a given actor is bounded by a constant, which is a reasonable assumption in practice, then the time complexity of *ConstructValidSchedule* is $O(If_if_o + |E|)$, where $G = (V, E)$ is the input SDF graph,

$$I = \sum_{A \in actors(G)} \mathbf{r}(A),$$

and f_i (f_o) is the maximum over all actors of the number of input (output) edges that are incident to any actor.

3.3 Scheduling to Minimize Buffer Usage

Although the algorithm presented above generates a valid class-S schedule, there are many other valid schedules that can be constructed. One desirable opti-

```
procedure ConstructValidSchedule (G, r)
        Initialize()
        repeat until ready is empty
            remove the actor A at the head of ready
            append queued(A) successive invocations of A to S
            scheduled(A) = scheduled(A) + queued(A)
            n = queued(A); queued(A) = 0
            for each input edge α of A
                b(α) = b(α) − (n × cns(α))
            end for
            for each output edge α of A
                b(α) = b(α) + (n × prd(α))
            end for
            for each output edge α of A
                r = r(snk(α)) − scheduled(snk(α))
                for each input edge β of snk(α)
                    r = min({r, ⌊b(β)/cns(β)⌋})
                end for
                if (r > queued(snk(α)))
                    append snk(α) to ready
                    queued(A) = r
                end if
            end for
        end repeat
        for each actor A in G
            if (scheduled(A) ≠ r(A))
                error: inconsistent graph
                exit
            end if
        end for
        output S
```

Figure 3.6. An algorithm for constructing a valid schedule for a connected SDF graph. The *initialize()* procedure is defined in Figure 3.7.

mization criterion is the amount buffer space that is required by the schedule. If we assume a model of buffering where there is one buffer on every edge, then the amount of buffer space required on each edge is given by the maximum number of tokens queued on that edge during one period of the schedule. It is easy to see that different valid schedules can have vastly different buffering requirements; for the graph in Figure 3.1, the schedule $AABBBCCC$ requires 11 spaces in all, while the schedule $BCABCABC$ requires 5 spaces. This section develops a heuristic that attempts to generate a schedule that minimizes buffering requirements.

Since an SDF schedule in general can have up to

$$I \equiv \sum_{A \in actors(G)} \mathbf{r}(A)$$

```
procedure Initialize()
    • define ready to be a queue of actors
    • define queued and scheduled to be vectors of non-negative
    integers indexed by the actors in G
    • define S to be a schedule; initialize S to be the null schedule.
    for each edge α
        b(α) = del(α)
    end for

    for each actor A in G
        r = r(A)
        for each input edge α of A
            r = min({r, ⌊del(α)/cns(α)⌋})
        end for
        if (r > 0) append A to the ready queue
            queued(A) = r
            scheduled(A) = 0
    ?   end if
    end for
```

Figure 3.7. The specification of the initialization routine that is used in Procedure ConstructValidSchedule of Figure 3.6.

actor appearances, algorithms that attempt to construct minimum-buffer schedules can take $O(I)$ steps or more to execute. Hence, these algorithms do not run in time that is polynomial *in the size of input SDF graph* (an algorithm would run in time polynomial in the size of the SDF graph if its running time were a polynomial function of N, the number of vertices in the graph, and $N \cdot \log(P)$ where P is the maximum of all of the produced/consumed parameters in the SDF graph; see Section 2.2). We will therefore require our algorithms for this purpose to run in time that is a polynomial function of $O(I)$, preferably a linear function. However, as the theorem below shows, the problem of computing a minimum buffer schedule for even an arbitrary homogenous SDF graph is NP-complete; hence, we have to rely on heuristics.

3.3.1. NP Completeness

Formally, the problem HSDF-MIN-BUFFER is defined as follows: Given an arbitrary homogenous SDF graph $G = (V, E)$, with arbitrary numbers of delays on each edge, and a positive integer K, is there a valid schedule for G that has a total buffering requirement of K or less, assuming that there is a buffer on every edge?

Definition 3.1: Let $D_T = \sum_{(u,v) \in E} del(u, v)$.

Observation 3.1: Any valid, minimal schedule for an HSDF graph has a buffering requirement B that satisfies $D_T \le B \le D_T + |E|$.

Proof: In an HSDF graph, each actor needs to be fired only once in any periodic schedule. If on any edge (u, v), $src((u, v))$ fires before $snk((u, v))$, then the maximum number of tokens queued is $del((u, v)) + 1$. If $snk((u, v))$ fires before $src((u, v))$ (provided $del((u, v)) > 0$) then the maximum number of tokens queued remains $del((u, v))$. In the worst schedule, the source actor fires before the sink actor on every edge; this increases the total amount of buffering by $|E|$. In the best conceivable schedule, the sink actor fires before the source on every edge (again, provided there is at least one delay on each edge), resulting in the lower bound of D_T.

Definition 3.2: The FEEDBACK ARC SET (FAS) problem is the following: given a directed graph $G = (V, A)$, and a positive integer K, does there exist a subset $B \subseteq A$ with $|B| \le K$, such that B contains at least one edge from every directed cycle in G? This problem is known to be NP-complete [Gare79].

Theorem 3.2: The HSDF-MIN-BUFFER problem is NP-complete.

Proof: The fact that HSDF-MIN-BUFFER is in NP is obvious. We reduce FEEDBACK ARC SET to this problem to show completeness. Let $G = (V, A)$, K be an arbitrary instance of the FAS problem. Let H be the HSDF graph constructed from G by reversing each edge in G and setting $del(e) = 1$ for each edge. We show that H contains a schedule having a buffering requirement $B \le K + |A| \Leftrightarrow$ there is a feedback arc set $A' \subseteq A$, $|A'| \le K$ in G. Observe that any permutation of the actors constitutes a valid schedule for H since there is a delay on every edge.

Assume that H contains such a schedule. This means that there are at most K edges in H where the source actor of the edge fires before the sink actor; on every other edge, the sink actor of the edge fires before the source actor. Construct a new delayless graph $H' = (V, E)$ in which an edge (u, v) from G is in E if and only if u fires before v in the schedule. An edge (v, u) is in E if (u, v) is an edge in G and v fires before u in the schedule. Hence, H is the same graph as G except that it has no delays and has reversed edges wherever the sink actor fires before the source actor in the schedule for H. These edges that are reversed in H' with respect to H are the same orientation as the corresponding edges in G. Hence, the set of edges in H' that have the reverse orientation with respect to edges in G, has cardinality at most K. By construction, it is clear that the schedule for H is also a valid schedule for H' since the schedule respects all of the precedence constraints in H'. Since H' is delayless, and has a valid schedule, it must be acyclic. Hence, reversing a subset of edges, with respect to G, where this subset has cardinality at most K results in an acyclic graph; this subset must be a solution to the FAS instance since the set has to include at least one edge from each directed cycle from G. Figure 3.8 shows an example.

Now suppose that G has a feedback arc set A' of cardinality less than or equal to K. In H, remove all of the edges corresponding to A' to form a new graph H'. The graph H' is now acyclic since at least one edge from every cycle has been removed. Construct a schedule for H' in reverse topological order; this ensures that sink actors on all of the edges fire before the source actors. This schedule is also valid for H. Moreover, in H, the schedule is such that on at most K edges can we have the situation that the source actor fires before the sink; hence, this schedule has a buffering requirement of at most $K + |A|$. ■

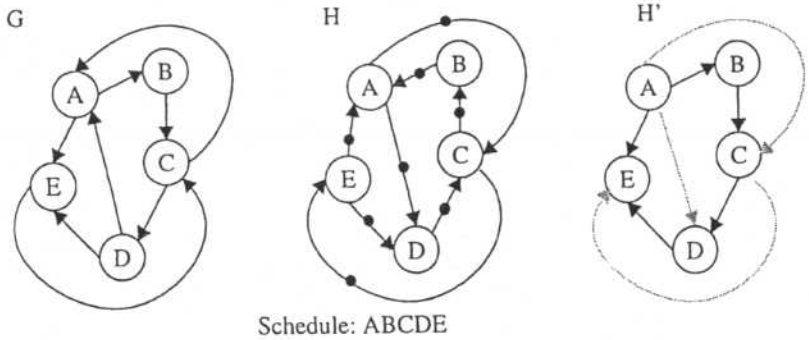


Figure 3.8: Example to illustrate proof of Theorem 3.2. A filled dot on an edge indicates a delay on that edge. The shaded edges in the H' graph show the edges that are reversed w.r.t G

3.3.2. Heuristic for Minimum Buffer Scheduling

Figure 3.9 shows a simple algorithm that constructs schedules that attempt to minimize buffer usage. A simpler variant of this algorithm has been used in both Gabriel and Ptolemy programming environments for a number of years. A similar algorithm is also given by Cubric and Panangaden in [Cubr93] where they establish its optimality for a restricted set of acyclic, delayless graphs. An actor in the algorithm is **deferrable** if any one of its output edges that is not a transitive edge has at least as many tokens as consumed by the sink actor on that edge in one firing.

```
Procedure min-buf-schedule (SDF Graph G=(V,E)):
F = {fireable actors};
D = {deferrable(F)};
while (F ≠ ϕ)
    if (F\D ≠ ϕ)
        // fire an actor from F\D
    else
        // fire an actor that increases total
        // number of tokens the least.
    end if
end while
```

Figure 3.9. A heuristic for generating a minimum buffer schedule.

Examples can easily be constructed where the algorithm fails to achieve the minimum buffer usage. However, in practice, the algorithm does quite well and produces schedules that are close to optimal even if not optimal. The algorithm can be made more sophisticated by making the actor selection procedure in the `else` condition less greedy; for example, by firing only actors that are in undirected cycles. However, it is an open problem as to whether such modifications will give an optimal algorithm for a broad class of graphs.

We close this section by proving a simple theorem about the lower bound on the amount of buffering required on any edge in an SDF graph. This result is a generalization of a result proved by researchers in the GRAPE project in [Ade94].

Theorem 3.3: For the 2-actor SDF graph depicted in Figure 3.10, the minimum amount of buffering required on the edge over all possible valid schedules is given by $a+b-c+d \bmod c$, where $c = gcd(a,b)$, if $0 \le d \le a+b-c$, and by d otherwise.

Proof: Clearly, the minimum buffer schedule for this graph is such that B is invoked whenever there are enough tokens on the edge to enable it. Also, the maximum number of tokens will be reached after a firing of A and before a firing of B. The theorem claims that $a+b-c+d \bmod c$ is the maximum number of tokens queued. To see this, assume that it is not; that in fact it is greater than this amount. Then it must be the case that the number of tokens is at least equal to $d \bmod c + ic$ where $ic > a+b-c$ (the change in the number of tokens on the edge is a multiple of c since if A has been fired x times and B has been fired y times, then the number of tokens on the edge is given by $xa - yb + d$ and this can be written as $ic + d \bmod c$.) Since $a+b-c$ is divisible by c, we get that $ic - a \ge b$. The maximum number of tokens always occurs after a firing of A and before a firing of B; hence, just before the maximum is reached, the number of tokens was $d \bmod c + ic - a \ge d \bmod c + b$ since $ic - a \ge b$. This shows that B was fireable at the previous step and contradicts the fact that we always fire it as soon as it becomes fireable.

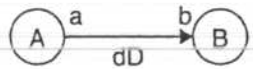


Figure 3.10. A simple 2-actor SDF graph.

It remains to show that this bound is tight and is in fact reached during the schedule. This can be seen by a state enumeration argument. Since A is fired b/c times and B is fired a/c times, the total number of states on the edge is $(a+b)/c$. Since any state that is reached is of the form $ic + d \bmod c$, and the number of states in $[0, a+b-c+d \bmod c]$ of this form is precisely $(a+b)/c$, we conclude that the state $a+b-c+d \bmod c$ is reachable and that the bound is tight (no state can be repeated since by definition a valid schedule for an SDF graph is one that returns the buffers to the initial state).

Suppose $d > a+b-c$. Initially, we can fire B enough times to get $d' = d \bmod b$ tokens on the arc. Now suppose that we were to exceed d tokens on the arc during the minimum buffer schedule. Then we would have $ic + d'$ tokens for some i, and $ic + d' > d$ or $ic > d - d'$ at some stage in the schedule. Since $d - d'$ is a multiple of b, and thus c, we have $ic + d' \geq d + c$. This number of tokens would have resulted after a firing of A; hence, before A was fired, we would have had $ic + d' - a \geq d + c - a$ tokens. Since $d > a+b-c$, $ic + d' - a > b$ and B was fireable when A was fired, contradicting the scheduling strategy. Hence, we cannot exceed d tokens under the minimum-buffer scheduling strategy. ■

4

Looped Schedules

4.1 Looped Schedule Terminology and Notation

This chapter begins by introducing some basic concepts and terminology pertaining to uniprocessor scheduling that will be used heavily throughout the remainder of the book.

Definition 4.1: Given an SDF graph G, a **schedule loop** is a parenthesized term of the form $(nT_1T_2...T_m)$, where n is a positive integer, and each T_i is either an actor in G or another schedule loop. The parenthesized term $(nT_1T_2...T_m)$ represents the successive repetition n times of the invocation sequence $T_1T_2...T_m$. If $L = (nT_1T_2...T_m)$ is a schedule loop, we say that n is the **iteration count** of L, each T_i is an **iterand** of L, and $T_1T_2...T_m$ constitutes the **body** of L. If the body of L is empty, that is if $m = 0$, we say that L is a **null** schedule loop; except where otherwise stated, we assume that all schedule loops are non-null. A **looped schedule** is a sequence $V_1V_2...V_k$, where each V_i is either an actor or a schedule loop. Since a looped schedule is usually executed repeatedly, we refer to each V_i as an **iterand** of the associated looped schedule.

When referring to a looped schedule, we often omit the "looped" qualification if it is understood from context; similarly, we may refer to a schedule loop simply as a **loop**. Given a looped schedule S, we refer to any contiguous sequence of actors and schedule loops in S (at any nesting depth) as a **subschedule** of S. For example, the schedules $(3AB)C$ and $(2B(3AB)C)A$ are both sub-

schedules of $A(2B(3AB)C)A(2B)$, whereas $(3AB)CA$ is not. By this definition, S is a subschedule of itself, and every schedule loop in S is a subschedule of S. If the same invocation sequence appears in more than one place in a looped schedule, we distinguish each instance as a separate subschedule. For example, in $(3A(2BC)D(2BC))$, there are two appearances of $(2BC)$, and these correspond to two distinct subschedules. In this case, the content of a subschedule is not sufficient to specify it — we must also specify the lexical position, as in "the second appearance of $(2BC)$." If S_0 is a subschedule of S, we say that S_0 is **contained in** S, and we say that S_0 is **nested in** S if S_0 is contained in S and $S_0 \neq S$.

We denote the set of actors that appear in a looped schedule S by $actors(S)$, and we denote the number of times that an actor A appears in S by $appearances(A, S)$; thus,

$$actors((2(2B)(5A))) = \{A, B\},$$

$$actors(X(2Y(3Z)X)) = \{X, Y, Z\},$$

$$appearances(C, (3CA)(4BC)) = 2,$$

$$appearances(A, (2ABAC)(3A)) = 3.$$

Formally, S is a **single appearance schedule** if

$$appearances(A, S) = 1 \ \forall A.$$

Given a looped schedule S and an actor A, we define $inv(A, S)$ to be the number of times that S invokes A. Similarly, if S_0 is a subschedule, we define $inv(S_0, S)$ to be the number of times that S invokes S_0. For example, if $S = A(2(3BA)C)BA(2B)$, then $inv(B, S) = 9$, $inv((3BA), S) = 2$, and $inv(\text{first appearance of } BA, S) = 6$. Also, we refer to the invocation sequence that a looped schedule S represents as the invocation sequence generated by S. For example, the invocation sequence generated by $S = A(2(3BA)C)BA(2B)$ is $ABABABACBABABACBABB$. When there is no ambiguity, we occasionally do not distinguish between a looped schedule and the invocation sequence that it generates.

A schedule loop is a **one-iteration** loop if its iteration count is 1. Although such loops are usually useless in the implementation of a schedule, they are useful for analyzing schedules, as will be apparent, for example, in Section 4.4. Since a one-iteration schedule loop generates the same invocation sequence as its body, replacing the loop by its body does not change the invocation sequence of an

enclosing schedule. Thus, given an arbitrary looped schedule S, if we select a one-iteration loop and replace it with its body, select a one-iteration loop in the resulting schedule and replace it with its body, and repeat this process until there are no one-iteration loops remaining, we will arrive at a new schedule S' that generates the same invocation sequence as S and contains no one-iteration loops. Thus, the following fact is obvious.

Fact 4.1: Given a looped schedule S, there exists a looped schedule S' that generates the same invocation sequence as S such that S' contains no one-iteration schedule loops, and

$$\forall (A \in actors(S)), appearances(A, S') = appearances(A, S).$$

Given a schedule S, an invocation I is said to be **part of** a subschedule S_0 if I occurs in an invocation of S_0. For example, in the schedule $AA(2AB)BB$, invocations A_3, A_4, B_1, and B_2 are part of the subschedule $(2AB)$, whereas A_1, A_2, B_3, and B_4 are not. Given an SDF graph G, an edge α in G, a looped schedule S for G, and a nonnegative integer i, we define $P(\alpha, i, S)$ to denote the number of invocations of $src(\alpha)$ that precede the ith invocation of $snk(\alpha)$ in S; and we define $T(\alpha, i, S)$ to denote the number of tokens on α just prior to the ith invocation of $snk(\alpha)$ in an execution of S. For example, consider the SDF graph in Figure 3.1 and let α denote the edge directed from B to C. Then $P(\alpha, 2, BC(2ABC)) = 2$, the number of invocations of B that precede invocation C_2 in the invocation sequence $BCABCABC$, and $T(\alpha, 2, BC(2ABC)) = 1$.

We will occasionally need to refer to the relative lexical positions of actors in a single appearance schedule. For this purpose, we define $position(X, S)$ to be the number of actors that lexically precede X in the single appearance schedule S. Observe that no ambiguity arises in this definition since we apply it only to single appearance schedules. For example, if $S = (2(3B)(5C))(7A)$, then $position(A, S) = 2$, $position(B, S) = 0$, and $position(C, S) = 1$. Formally, we define the **lexical ordering** of a single appearance schedule S, denoted $lexorder(S)$, to be the sequence of actors $(A_1, A_2, \ldots, A_n)$ where $\{A_1, A_2, \ldots, A_n\} = actors(S)$ and $position(A_i, S) = i - 1$ for each i. Thus, $lexorder((2(3B)(5C))(7A)) = (B, C, A)$. We will apply the following obvious fact about lexical orderings.

Fact 4.2: If S is a valid single appearance schedule for a delayless SDF graph[1], then whenever X is an ancestor of Y, we have

$$position(X, S) < position(Y, S).$$

4.2 Buffering Model

Given a looped schedule S for an SDF graph G, we define the **buffer memory** required by S, denoted $buffer_memory(S)$, to be the number of storage units required to implement the buffering for S if each buffer is mapped to a separate contiguous block of memory. We assume that every token occupies one storage unit. Quantitatively, if $max_tokens(\alpha, S)$ denotes the maximum number of tokens that are simultaneously queued on edge α during an execution of the schedule S, we have that

$$buffer_memory(S) = \sum_{\alpha \in edges(G)} max_tokens(\alpha, S).$$

We define the **buffer memory requirement** of a schedule S, as $buffer_memory(S)$. In Figure 3.1, if $S = BC(2ABC)$, then $max_tokens(A \rightarrow B, S) = 4$, $max_tokens(B \rightarrow C, S) = 1$, and $buffer_memory(S) = 4 + 1 = 5$.

The amount of memory required for buffering may vary greatly between different schedules. For example, the schedule $(9A)(12B)(12C)(8D)$ has a buffer memory requirement of $36 + 12 + 24 = 72$, and the schedule $(3(3A)(4B))(4(3C)(2D))$ has a buffer memory requirement of $12 + 12 + 6 = 30$ for the graph in Figure 4.1.

In the model of buffering implied by the "buffer memory requirement" measure, each buffer is mapped to a contiguous and independent block of memory. This model is convenient and natural for code generation, and it is the model used, for example, in the SDF-based code generation environments described in [Ho88b, Pino94, Ritz92]. However, perfectly valid target programs can be generated without these restrictions. For example, another model of buffering (for a chain-structured graph) is to use a shared buffer of size

Figure 4.1. A chain-structured SDF graph.

[1] Note that a consistent, delayless SDF graph is necessarily acyclic.

$$max(\{\mathbf{q}(N_i) \times prd(N_i) \mid 1 \le i < K\})$$

which gives the maximum amount of data transferred on any edge in one execution of the single appearance schedule $(\mathbf{q}(N_1)N_1)(\mathbf{q}(N_2)N_2)\ldots(\mathbf{q}(N_K)N_K)$, where K is the number of vertices in the graph. Assuming that there are no delays on the graph edges, it can be shown that via proper management of pointers, such a buffer suffices. For the example graph 4.1, this would imply a buffering requirement of 36 since on edge AB, 36 samples are exchanged in the schedule $(9A)(12B)(12C)(8D)$, and this is the maximum over all edges. Moreover, the implementation of this schedule using a shared buffer would be much simpler than the implementation of a more complicated nested schedule.

But there are two problems with buffer-sharing that prevent its use as the model for evaluating the buffering cost of single appearance schedules (even for chain-structured graphs; the general case is more complicated). Consider the graph in Figure 4.2. The shared-buffer cost for the schedule $S' = (A)(50B)(100C)(4D)$ for this graph is given by

$$max(\{1 \times 50, 50 \times 100, 100 \times 50, 4 \times 25\}) = 5000.$$

However, with a buffering model where we have a buffer on each edge, the schedule $A(50B(2C))(4D)$ requires total buffering of only 250 units. Of-course, we could attempt sharing buffers in this nested looped schedule as well, but the implementation of such sharing could be awkward.

Consider also the effect of having delays on the edges. In the model where we have a buffer on every edge, having delays does not affect the ease of implementation. For example, if we introduce d delays on edge BC in the graph in Figure 4.2, then we merely augment the amount of buffering required on that edge by d. This is fairly straightforward to implement. On the other hand, having delays in the shared buffer model causes complications because there is often no

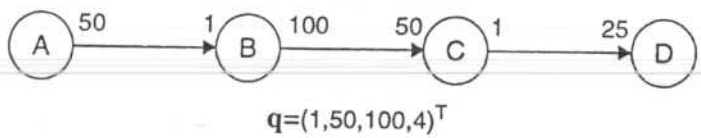


Figure 4.2. Example to illustrate the inefficiency of using shared buffers.

logical place in the buffer to store the initial samples since the entire buffer might be written over by the time we reach the actor that consumes the delays. For instance, consider the graph in Figure 4.3. This graph is an SDF abstraction of a possible system to do sample rate conversion between CD rates and DAT rates; see Section 6.3.1. for more details. The repetitions vector for this graph is given by $(147, 49, 28, 32, 160)^T$. Suppose that we were to use the shared-buffer implementation for schedule S'. We find that we need a buffer of size 224. After all of the invocations of A have been fired, the first 147 locations of the buffer are filled. Since B writes more samples than it reads, it starts writing at location 148 and writes 196 samples. When C begins execution, it starts reading from location 148 and starts writing from location 120 (120 = (148+196) mod 224). Actor C then writes 224 samples into the buffer. When D is invoked, it starts reading from location 120. Hence, if there were a delay on edge CD for instance, the logical thing to do would be to have a buffer of size 225 (meaning that D would start reading from location 119) and place the delay in location 119. However, location 119 would have been written over by A; hence, it is not a safe location. This shows that handling delays in the shared buffer model can be quite awkward, and would probably involve copying over data from a "delay" buffer of some sort.

For these reasons, this book focuses on the buffering model associated with the "buffer memory requirement" measure. An extension of these techniques to combine the above simple model of buffer sharing with the non-shared model was presented in [Murt94a] for a more restricted class of graphs.

4.3 Clustering SDF Subgraphs

The fundamentals of SDF were introduced in terms of connected SDF graphs. This section extends some basic principles of SDF to non-connected SDF graphs. We begin with a definition.

4.3.1. Projections

Definition 4.2: Suppose that S is a looped schedule for an SDF graph G and $Z \subseteq actors(G)$. If we remove from S all actors that are not in Z and then we repeatedly remove all null loops until no null loops remain, we obtain another

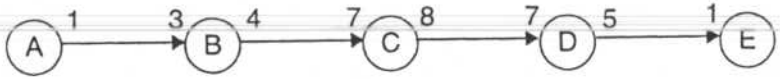


Figure 4.3. Example to illustrate the difficulty of using shared buffers with delays.

looped schedule, which we call the **projection** of S onto Z, and which we denote by $projection(S, Z)$. For example,

$$projection((2(2B)(5A)), \{A, C\}) = (2(5A)),$$

and $projection((5C), \{A, B\})$ is the null schedule. If G' is a subgraph of G, we define $projection(S, G') \equiv projection(S, actors(G'))$.

We will use the following fact, which follows immediately from Definition 4.2 and the definitions introduced in the previous section.

Fact 4.3: If S and S' are valid looped schedules for an SDF graph G, α is an edge in G, and i is a positive integer such that $1 \le i \le inv(snk(\alpha), S)$ and $1 \le i \le inv(snk(\alpha), S')$, then

(a). $(P(\alpha, i, S) = P(\alpha, i, S')) \Leftrightarrow (T(\alpha, i, S) = T(\alpha, i, S'))$;

(b). $P(\alpha, i, S) = P(\alpha, i, projection(S, \{src(\alpha), snk(\alpha)\}))$; and

(c). $max_tokens(\alpha, S) = max(\{T(\alpha, i, S) | (1 \le i \le inv(snk(\alpha), S))\})$.

The projection of an admissible schedule S onto a subset of actors Z fully specifies the sequence of token populations occurring on each edge in the corresponding subgraph. More precisely, for any actor $A \in Z$, any positive integer i such that $1 \le i \le inv(A, S)$, and any input edge α of A contained in $subgraph(Z)$, the number of tokens queued on α just prior to the ith invocation of A in S equals the number of tokens queued on α just prior to the ith invocation of A in an execution of $projection(S, Z)$. Thus, we have the following fact.

Fact 4.4: If S is a schedule for an SDF graph G, G' is a subgraph of G, and α is an edge in G', then

(a) If S is valid (periodic) then $projection(S, G')$ is a valid (periodic) schedule for G';

(b) S terminates on α implies that $projection(S, G')$ terminates on α; and

(c) If S is valid, then

$$max_tokens(\alpha, projection(S, G')) = max_tokens(\alpha, S).$$

4.3.2. Blocking Vectors

The concept of blocking factor does not apply directly to SDF graphs that are not connected. For example in Figure 4.4 the minimal numbers of repetitions for a periodic schedule is given by $\rho(A, B, C, D) = (1, 1, 1, 1)^T$ [2]. The schedule $A(2C)B(2D)$ is a valid schedule for this example, but this schedule corresponds to a blocking factor of 1 for $subgraph(\{A, B\})$ and a blocking factor of 2 for

$subgraph(\{C, D\})$ — there is no single *scalar* blocking factor associated with $A(2C)B(2D)$.

Now suppose that S is a valid schedule for an arbitrary SDF graph G. By Fact 4.4, for each connected component C of G, we have that $projection(S, C)$ is a valid schedule for $subgraph(C, G)$. Thus, associated with S, there is a vector of positive integers $\mathbf{J}_S$, indexed by the connected components of G, such that for each connected component C of G,

$$A \in C \Rightarrow inv(A, S) = \mathbf{J}_S(C)\mathbf{q}_{subgraph(C)}(A). \tag{4.1}$$

We call $\mathbf{J}_S$ the **blocking vector** of S. For example, if $S = A(2C)B(2D)$ for Figure 4.4, then $\mathbf{J}_S(\{A, B\}) = 1$, and $\mathbf{J}_S(\{C, D\}) = 2$. On the other hand, if G is connected, then $\mathbf{J}_S$ has only one component, which is the blocking factor of S, $J(S)$. We refer to any vector of positive integers indexed by the connected components of G as a blocking vector for G.

It is often convenient to view a part of an SDF graph as a subsystem that is invoked as a single unit. The invocation of a subsystem corresponds to invoking a minimal valid schedule for the associated subgraph. If this subgraph is connected, its repetitions vector gives the minimum number of invocations required for a periodic schedule. However, if the subgraph is not connected, then the minimum number of invocations involved in a periodic schedule is not necessarily obtained by concatenating the repetitions vectors associated with the connected components of the subgraph. This is because the full SDF graph may contain connections between the non-connected components of the subgraph.

For example, let G denote the SDF graph in Figure 4.5(a) and consider the subsystem $\Omega = subgraph(\{A, B, C, D\})$ in this graph. It is easily verified that $\mathbf{q}_G(A, B, C, D, E) = (2, 2, 4, 4, 1)^T$. Thus in a periodic schedule, the actors in $subgraph(\{C, D\})$ must be invoked twice as frequently as those in

Figure 4.4. A simple non-connected SDF graph.

2. Note that this vector is not a repetitions vector, and thus it is not represented by $\mathbf{q}$, because the associated graph is not connected. By definition, only connected SDF graphs have repetitions vectors.

$subgraph(\{A, B\})$. We see that for a periodic schedule, the minimum numbers of repetitions for $subgraph(\{A, B, C, D\})$ *as a subgraph of* the original graph are given by $\rho(A, B, C, D) = (1, 1, 2, 2)^T$, which can be obtained by dividing each corresponding component in $\mathbf{q}_G$ by $gcd(\{\mathbf{q}_G(A), \mathbf{q}_G(B), \mathbf{q}_G(C), \mathbf{q}_G(D)\}) = 2$.

On the other hand, concatenating the repetitions vectors of $subgraph(\{A, B\})$ and $subgraph(\{C, D\})$ yields the repetition counts $\rho'(A, B, C, D) = (1, 1, 1, 1)^T$. However, repeatedly invoking the subsystem with these relative repetition rates can never lead to a periodic schedule for G. We have motivated the following definition.

Definition 4.3: Let G be a connected SDF graph, suppose that Z is a subset of $actors(G)$, and let $R = subgraph(Z)$. We define $q_G(Z) \equiv gcd(\{\mathbf{q}_G(A) | A \in Z\})$, and we define $\mathbf{q}_{R/G}$ to be the vector of positive integers indexed by the members of Z that is defined by $\mathbf{q}_{R/G}(A) \equiv \mathbf{q}_G(A)/q_G(Z)$, for each $A \in Z$. We can view $q_G(Z)$ as *the number of times a minimal periodic schedule for G invokes the subgraph R*, and accordingly, we refer to we refer the quantity $q_G(Z)$ as the **repetition count** of Z, and we refer to $\mathbf{q}_{R/G}$ as the **repetitions vector of R as a subgraph of G**. For example, in Figure 4.5(a), if $R = subgraph(A, B, C, D)$, then $q_G(actors(R)) = 2$, and $\mathbf{q}_{R/G} = \mathbf{q}_{R/G}(A, B, C, D) = (1, 1, 2, 2)^T$.

For convenience, if A and B are adjacent actors, then we define $\rho_G(\{A, B\}) \equiv q_G(\{A, B\})$; that is, $\rho_G(\{A, B\}) = gcd(\{\mathbf{q}_G(A), \mathbf{q}_G(B)\})$. If G is understood from the context, then we may drop the subscript and simply write $\rho(\{A, B\})$.

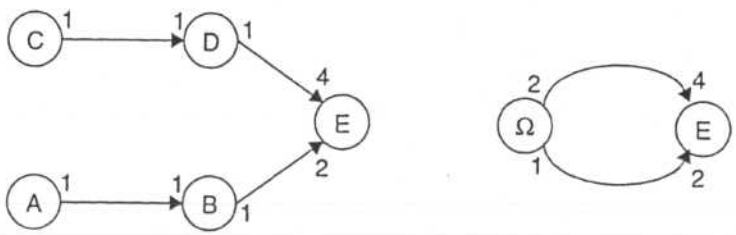


Figure 4.5. An example of clustering a subgraph in an SDF graph.

The following fact establishes that for a connected SDF subgraph, its repetitions vector is the repetitions vector of itself as a subgraph of the enclosing graph.

Fact 4.5: If G is a connected SDF graph and R is a connected subgraph of G, then $\mathbf{q}_{R/G} = \mathbf{q}_R$. Thus, for a connected subgraph R, for each $A \in actors(R)$, $\mathbf{q}_G(A) = q_G(actors(R))\mathbf{q}_R(A)$.

Proof: Let S be any periodic schedule for G of unit blocking factor, and let $S' = projection(S, R)$. Then from Fact 4.4 and Fact 3.5, for all $A \in actors(R)$, we have $\mathbf{q}_G(A) = J(S')\mathbf{q}_R(A)$. From Fact 3.4, we know that the components of $\mathbf{q}_R$ are coprime, and it follows that

$$J(S') = gcd(\{\mathbf{q}_G(A') | A' \in actors(R)\}) = q_G(actors(R)).$$

Thus, for each actor A in R, $\mathbf{q}_R(A) = \mathbf{q}_G(A)/q_G(actors(R)) = \mathbf{q}_{R/G}(A)$. ■

For example, in Figure 4.5(a), let $R = subgraph(\{A, B\})$. We have $\mathbf{q}_G(A, B, C, D, E) = (2, 2, 4, 4, 1)^T$, $\mathbf{q}_R(A, B) = (1, 1)^T$, and from Definition 4.3, $q_G(actors(R)) = gcd(2, 2) = 2$ and $\mathbf{q}_{R/G}(A, B) = (2, 2)^T/2 = (1, 1)^T$. As Fact 4.5 assures us, $\mathbf{q}_R = \mathbf{q}_{R/G}$.

4.3.3. Clustering

We will extensively apply the concept of "clustering" a subgraph in an SDF graph. Given a connected, consistent SDF graph $G = (V, E)$, a subset $Z \subseteq V$, and an actor $\Omega \notin V$, **clustering** Z **into** Ω means generating the new SDF graph (V', E') such that $V' = V - Z + \{\Omega\}$ and

$$E' = E - (\{e | (src(e) \in Z) \textbf{ or } (snk(e) \in Z)\}) + E^*,$$

where E^* is a "modification" of the set of edges that connect actors in Z to actors outside of Z. If for each $e \in E$ such that $src(e) \in Z$ and $snk(e) \notin Z$, we define e' by

$$src(e') = \Omega,\ snk(e') = snk(e),$$

$$del(e') = del(e),\ prd(e') = prd(e) \times (\mathbf{q}_G(src(e))/q_G(Z)), \text{ and}$$

$$cns(e') = cns(e);$$

and similarly, for each $e \in E$ such that $snk(e) \in Z$ and $src(e) \notin Z$, we define e' by

$$src(e') = src(e),\ snk(e') = \Omega$$

$$del(e') = del(e),\ prd(e') = prd(e), \text{ and}$$

$$cns(e') = cns(e) \times (\mathbf{q}_G snk(e)/q_G(Z)),$$

then, we can specify E^* by

$$E^* = \{e' | (src(e) \in Z \textbf{ and } snk(e) \notin Z) \textbf{ or } (snk(e) \in Z \textbf{ and } src(e) \notin Z)\}.$$

For each $e' \in E^*$, we say that e' **corresponds to** e and vice versa (e corresponds to e'). The graph that results from clustering Z into Ω in G is denoted $clust(Z, G, \Omega)$, or simply $clust(Z, G)$. Intuitively, an invocation of Ω in $clust(Z, G, \Omega)$ corresponds to an invocation of a minimal valid schedule for $subgraph(Z)$ in G. We say that Z is **clusterable** if $clust(Z, G)$ is consistent, and if G is acyclic, we say that Z **introduces a cycle** if $clust(Z, G)$ contains one or more cycles. Figure 4.6 gives an example of clustering. Here, edge (D, Ω) corresponds to (D, C) (and vice versa), and (Ω, A) corresponds to (B, A).

Fact 4.6: Suppose G is an SDF graph, R is a subgraph of G, G' is the SDF graph that results from clustering R into the hierarchical actor Ω, S' is a valid schedule for G', and S_R is a valid schedule for R such that for each actor A in R, $inv(A, S_R) = \mathbf{q}_{R/G}(A)$. Let $\hat{S}$ denote the schedule that results from replacing each instance of Ω in S' with S_R. Then $\hat{S}$ is a valid schedule for G.

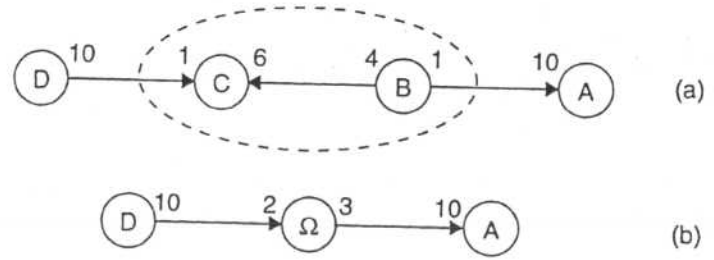


Figure 4.6. An example of clustering. In (b), we have $clust(\{B, C\}, G, \Omega)$, where G denotes the SDF graph in (a). Here, $\mathbf{q}_G(A, B, C, D) = (3, 30, 20, 2)$, and thus, $\rho_G(\{B, C\}) = 10$.

As a simple example, consider Figure 4.5 again. Now, $(2\Omega)E$ is a valid schedule for the SDF graph in Figure 4.5(b), and $S = AB(2CD)$ is a valid schedule for $R = subgraph(\{A, B, C, D\})$ such that $(inv(A', S) = \mathbf{q}_{R/G}(A')), \forall A'$. Thus Fact 4.6 guarantees that $(2AB(2CD))E$ is a valid schedule for Figure 4.5(a).

Proof of Fact 4.6: Given a schedule S and an SDF edge α, we define

$$\Delta(\alpha, S) \equiv (inv(src(\alpha), S) \times prd(\alpha)) - (inv(snk(\alpha), S) \times cns(\alpha)). \qquad (4.2)$$

Then S is periodic if and only if it invokes each actor and $(\Delta(\alpha, S) = 0)\forall\alpha$.

We can decompose S' into $s_1\Omega s_2\Omega\ldots s_{k-1}\Omega s_k$, where each s_j denotes the sequence of invocations between the $(j-1)$th and jth invocations of Ω. Then $\hat{S} = s_1 S_R s_2 S_R \ldots S_R s_k$.

First, suppose that β is an edge in G such that $src(\beta), snk(\beta) \notin actors(R)$. Then S_R contains no occurrences of $src(\beta)$ nor $snk(\beta)$, so $P(\beta, i, \hat{S}) = P(\beta, i, S')$ for any invocation number i of $snk(\beta)$. Thus, since S' is admissible, $\hat{S}$ does not terminate on β. Also, $\Delta(\beta, \hat{S}) = \Delta(\beta, s_1 s_2 \ldots s_k) = \Delta(\beta, S') = 0$, since S' is periodic.

If $src(\beta), snk(\beta) \in actors(R)$, then none of the s_j's contain any occurrences of $src(\beta)$ or $snk(\beta)$. Thus, for any i, $P(\beta, i, \hat{S}) = P(\beta, i, \hat{S}')$, where $\hat{S}' = S_R S_R \ldots S_R$ denotes $\hat{S}$ with all of the s_j's removed. Since $\hat{S}'$ consists of successive invocations of a valid schedule, it follows that $\hat{S}$ does not terminate on β, and $\Delta(\beta, \hat{S}) = 0$.

Now suppose that $src(\beta) \in actors(R)$ and $snk(\beta) \notin actors(R)$. Then corresponding to β, there is an edge β' in G', such that $src(\beta') = \Omega$, $snk(\beta') = snk(\beta)$, $prd(\beta') = \mathbf{q}_{R/G}(src(\beta)) \times prd(\beta)$, and $cns(\beta') = cns(\beta)$. Now each invocation of S_R produces

$$inv(src(\beta), S_R)prd(\beta) = \mathbf{q}_{R/G}(src(\beta))prd(\beta) = prd(\beta')$$

tokens onto β. Since $cns(\beta') = cns(\beta)$ and S' is a valid schedule, it follows that $\Delta(\beta, \hat{S}) = 0$, and $\hat{S}$ does not terminate on β.

Similarly, if $src(\beta) \notin actors(R)$ and $snk(\beta) \in actors(R)$, we see that each invocation of S_R consumes the same number of tokens from β as Ω consumes from the corresponding edge in G', and thus $\Delta(\beta, \hat{S}) = 0$ and $\hat{S}$ does not terminate on β.

We conclude that $\hat{S}$ does not terminate on any edge in G, and thus, $\hat{S}$ is admissible. Furthermore, $\Delta(\alpha, \hat{S}) = 0$ for each edge α in G, and since S' and S_R are both periodic schedules, it is easily verified that $\hat{S}$ invokes each actor in G at least once, so we conclude that $\hat{S}$ is a periodic schedule. ■

The following fact relates the repetitions vector of an SDF graph obtained by clustering a subgraph to the repetitions vector of the original graph.

Fact 4.7: If G is a connected SDF graph, $Z \subseteq actors(G)$, and G' is the SDF graph obtained from G by clustering $subgraph(Z)$ into the actor Ω, then $\mathbf{q}_{G'}(\Omega) = q_G(Z)$, and $\forall A \in (actors(G) - Z)$, $\mathbf{q}_{G'}(A) = \mathbf{q}_G(A)$.

Proof: Let $\mathbf{q}'$ denote the vector that we claim is the repetitions vector for G', and recall from Fact 3.4 that $\mathbf{q}' = \mathbf{q}_{G'}$ if and only if $\mathbf{q}'$ satisfies the balance equations for G' and the components of $\mathbf{q}'$ are coprime. From the definition of *clustering*, it can easily be verified that $\mathbf{q}'$ satisfies the balance equations for G'. Furthermore, from Fact 3.4, no positive integer greater than 1 can divide all members of

$$(\{\mathbf{q}_G(A) | (A \in actors(G) - Z)\} \cup \{gcd(\{\mathbf{q}_G(A) | (A \in Z)\})\}).$$

Since $\mathbf{q}'(\Omega) = gcd(\{\mathbf{q}_G(A) | (A \in Z)\})$, it follows that the components of $\mathbf{q}'$ are coprime. ■

Fact 4.6 and Fact 4.7 imply that for scheduling purposes, a cluster in a connected SDF graph can be viewed as monolithic from the outside or as an SDF graph (possibly non-connected) from the inside, and that the SDF parameters of the monolith and the repetitions vector of the graph that it is contained in can be formally bound to the repetitions vector of the original SDF graph.

We conclude this section with the following consequence of Fact 4.7.

Fact 4.8: If G is a connected SDF graph, $Z \subseteq actors(G)$, and G' is the SDF graph obtained from G by clustering $subgraph(Z)$ into the actor Ω, then for each edge e in G', $TNSE(e, G') = TNSE(e', G)$, where e' is the edge in G that corresponds to e.

Proof: Let e be an edge in G', and let e' denote the corresponding edge in G. By definition of clustering, either $src(e) \neq \Omega$ or $snk(e) \neq \Omega$. If $src(e) \neq \Omega$, then $src(e) = src(e'), prd(e') = prd(e)$ and from Fact 4.7,

$$\mathbf{q}_{G'}(src(e)) = \mathbf{q}_G(src(e')).$$

Thus, $(src(e) \neq \Omega)$ implies that

$$TNSE(e, G') = \mathbf{q}_{G'}(src(e))prd(e) = \mathbf{q}_G(src(e'))prd(e') = TNSE(e', G).$$

A similar argument can be constructed for the case $snk(e) \neq \Omega$. ■

The concept of a cluster in a graph has been defined in and applied in many different contexts. In VLSI circuits, for example, a "cluster" is informally defined as a particularly dense or complex subcircuit, and the problem of detecting such clusters has been addressed to partition a circuit so that the number of connections crossing the partition are minimized [Garb90]. In multiprocessor scheduling, clustering is commonly used to group subsets of dataflow actors that are to be scheduled on the same processor [Gera92]. A third example arises in the context of dataflow/von Neumann hybrid architectures, which allow collections of data flow actors, called threads, to execute sequentially under the control of a program counter, while the invocation of threads is carried out in a data-driven manner. Thus, the computation within a thread is performed in a von Neumann style, while the threads themselves are sequenced in a dataflow style. When compiling for a hybrid dataflow/von Neumann machine, clustering can be used to construct coarse-grain threads from a fine-grain dataflow representation of the program [Najj92].

4.4 Factoring Schedule Loops

This section shows that in a single appearance schedule, common terms from the iteration counts of inner loops can be "factored" into the iteration counts of the enclosing loops. An important practical advantage of factoring is that it may significantly reduce the buffer memory requirement.

For example, consider the SDF graph in Figure 4.7. Here, $\mathbf{q}(A, B, C, D) = (100, 100, 10, 1)^T$, and one valid single appearance schedule for this graph is $(100A)(100B)(10C)D$. With this schedule, prior to each invocation of C, 100 tokens are queued on each of the input edges of C, and a maximum of

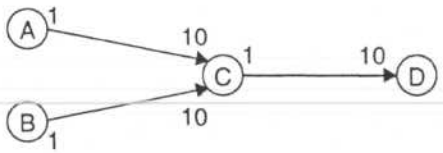


Figure 4.7. An SDF graph used to illustrate the factoring of loops.

10 tokens are queued on the input edge of D. Thus 210 units of storage are required to implement the buffering of tokens for this schedule

Now observe that this schedule generates the same invocation sequence as $(1(100A)(100B)(10C))D$. The main result developed in this section allows us to factor the common divisor of 10 in the iteration counts of the three inner loops into the iteration count of the outer loop. This yields the new single appearance schedule $(10(10A)(10B)C)D$, for which at most 10 tokens simultaneously reside on each edge. Thus, this factoring application has reduced the buffer memory requirement by a factor of 7.

There is, however, a trade-off involved in factoring. For example, the schedule $(100A)(100B)(10C)D$ requires 3 loop initiations per schedule period, while the factored schedule $(10(10A)(10B)C)D$ requires 21. Thus, the run-time cost of starting loops — usually, initializing the loop indices — has increased by the same factor by which the buffer memory requirement has decreased. However, for programmable digital signal processors, the loop-startup overhead is normally much smaller than the penalty that is paid when the memory requirement exceeds the on-chip limits. Unfortunately, we cannot in general perform the reverse of the factoring transformation; that is, moving a factor from the iteration count of an outer loop to the iteration counts of the inner loops. This *reverse factoring* transformation might be desirable in situations where minimizing the buffer memory requirement is not critical.

Figure 4.10 shows a simple SDF graph that can be used to demonstrate that unlike the factoring transformation, reverse factoring does not necessarily preserve the admissibility of a valid single appearance schedule. It is easily verified that $(10AB)$ is a valid single appearance schedule (with blocking factor 10) for this graph, while the reverse-factored derivative $(10A)(10B)$ terminates on the

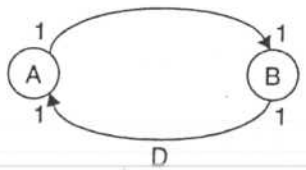


Figure 4.8. An example used to illustrate that reverse factoring is not always valid for single appearance schedules.

edge $B \rightarrow A$ at the second invocation of A. Further exploration of reverse factoring is beyond the scope of this book.

The factoring transformation is closely related to the **loop fusion** transformation, which has been used for decades in compilers for procedural languages. In the basic version of this transformation, two adjacent loops having the same iteration count are merged into a single loop by concatenating the bodies. It is well-known that loop fusion can reduce a program's memory requirements [Wolf89] just as the factoring transformation that is presented in this section does. Also, loop fusion has been found to increase data locality, and hence to improve the exploitation of memory hierarchies [AbuS81]. In compilers for procedural languages, tests for the validity of loop fusion include analysis of array subscripts to determine whether or not for each iteration n of the (lexically) second loop, this iteration depends only on iterations $1, 2, \ldots n$ of the first loop [Zima90]. These tests are difficult to perform comprehensively due to the complexity of exact subscript analysis [Bane88], and due to complications such as data-dependent subscript values, conditional branches inside one or more of the loops, and input/output statements. This section shows that for single appearance schedules of SDF graphs, these complications do not arise, and loop fusion, as well as the more general factoring transformation, is always a valid transformation. Thus, loop fusion is an additional example of the increased compile-time predictability that can be gained when restricting the computational model to SDF.

We will apply the following simple number-theoretic fact in the development of this section.

Fact 4.9: Suppose that x is a nonnegative integer, and a, b, and y are positive integers such that $x + ya \geq yb$. Then $\forall(i \in \{0, 1, \ldots, y\})$, $x + ia \geq ib$.

Proof: Suppose that $i \in \{0, 1, \ldots, y\}$, and first suppose that $a \leq b$. Then,

$$x + ya \geq yb \Rightarrow x \geq (b-a)y \Rightarrow x \geq (b-a)i \Rightarrow x + ia \geq ib.$$

While, on the other hand if $a > b$, then $(b-a)i \leq 0$, and since $x \geq 0$, it follows that $(b-a)i \leq x$, and thus $x + ia \geq ib$. ■

The following lemma establishes similarities between two looped schedules S and S' for a consistent SDF graph, where S' is obtained by replacing some subschedule S_0 in S with another schedule S_0' that invokes each actor the same number of times as S_0. For an SDF edge α, Lemma 4.1(a) states that if the

mth invocation of $snk(\alpha)$ is not part of S_0' in S', then the number of invocations of $src(\alpha)$ that precede the mth invocation of $snk(\alpha)$ in S' equals the number of invocations of $src(\alpha)$ that precede the mth invocation of $snk(\alpha)$ in S; and Lemma 4.1(b) asserts the same conclusion whenever α is not contained in the subgraph associated with the set of actors invoked by S_0. We first state and prove the lemma, and present a corollary, and then we illustrate with an example.

Lemma 4.1: Suppose that G is a consistent SDF graph, S is a looped schedule for G, and S_0 is a subschedule of S. Suppose also that S_0' is any looped schedule such that $actors(S_0') = actors(S_0)$, and $inv(A, S_0') = inv(A, S_0)$, $\forall(A \in actors(S_0))$. Let S' denote the schedule obtained by replacing S_0 with S_0' in S. Then for any actor $N \in actors(G)$, any positive integer m such that $1 \leq m \leq inv(N, S)$, and any input edge α of N, we have

(a). If invocation N_m is not part of S_0' in S', then $P(\alpha, m, S') = P(\alpha, m, S)$; and

(b). If α is not contained in $subgraph(actors((S_0), G))$, then $P(\alpha, m, S') = P(\alpha, m, S)$.

Proof: The sequence of invocations in S can be decomposed into $s_1 b_1 s_2 b_2 \ldots b_n s_{n+1}$, where b_j denotes the sequence of invocations associated with the jth invocation of subschedule S_0, and s_j is the sequence of invocations between the $(j-1)$th and jth invocations of S_0. Since S' is derived by rearranging the invocations in S_0, we can express S' similarly as $s_1 b_1' s_2 b_2' \ldots b_n' s_{n+1}$, where b_j' corresponds to the jth invocation of S_0' in S'.

For the proof of part (a), observe that N_m is part of some s_j, say s_k, in S'. Then $k-1$ invocations of S_0' precede N_m in S', and since for all j, $inv(N, b_j') = inv(N, b_j)$ we have that in S, N_m is also part of s_k. It follows that

$$P(\alpha, m, S) =$$
$$P(\alpha, m - (k-1)\,inv(N, S_0), s_1 s_2 \ldots s_k) + (k-1)\,inv(src(\alpha), S_0),$$

and

$$P(\alpha, m, S') =$$
$$P(\alpha, m - (k-1)\,inv(N, S_0'), s_1 s_2 \ldots s_k) + (k-1)\,inv(src(\alpha), S_0').$$

But, by assumption, $inv(src(\alpha), S_0) = inv(src(\alpha), S_0')$, and $inv(N, S_0) = inv(N, S_0')$, so $P(\alpha, m, S) = P(\alpha, m, S')$, and the proof of part (a) is complete.

For the proof of part (b), first observe that if in S, N_m is part of one of the s_j's, then from part (a), we have immediately that $P(\alpha, m, S) = P(\alpha, m, S')$. On the other hand, if N_m is part of one of the b_j's, say b_p, in S, then $inv(N, S_0) = inv(N, S_0')$ implies that in S', N_m is part of b_p'. Also, by assumption α is not in $subgraph(actors((S_0), G))$, so since N_m is part of b_p, $N = snk(\alpha)$ is contained in $actors(S_0)$, and thus $src(\alpha) \notin actors(S_0)$. It follows that $P(\alpha, m, S) = inv(src(\alpha), s_1 s_2 \ldots s_p) = P(\alpha, m, S')$, and the proof of part (b) is complete. ■

The following corollary follows immediately from Lemma 4.1. It implies that under the assumptions of Lemma 4.1 together with the additional assumption that S is admissible, if an invocation is not part of S_0' in S', then S' cannot terminate at that invocation, and if α is not contained in the subgraph associated with $actors(S_0)$, then S' cannot terminate on α.

Corollary 4.1: Assume the hypotheses of Lemma 4.1 with the additional assumption that S is admissible. Then

(a). If invocation N_m is not part of S_0' in S', then S' does not terminate on α at N_m.

(b). If α is not contained in $subgraph(actors(S_0), G)$, then S' does not terminate on α.

Consider the example in Figure 4.9. Here, each d_j represents the number of delays on the corresponding edge, and the repetitions vector is given by $\mathbf{q}(A, B, C, D, E) = (2, 4, 4, 1, 1)^T$. Suppose that the d_j's are such that $D(2A(2BC))E$ is an admissible schedule. Then Corollary 4.1(b) — with $S_0 = A(2BC)$ and $S_0' = BCABC$ — guarantees that the schedule $D(2BCABC)E$ does not terminate on the edges $B \to D$, $D \to A$, $E \to A$, or

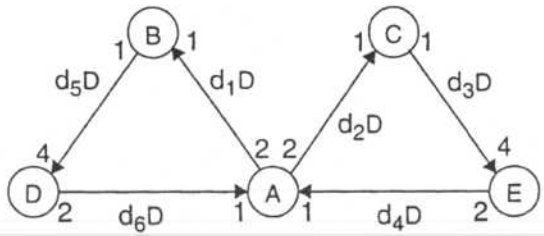


Figure 4.9. An example used to illustrate Lemma 4.1.

$C \to E$; and Corollary 4.1(a) insures that this schedule does not terminate at the invocation of D_1 or E_1.

The following lemma establishes a simple sufficient condition for valid application of the loop fusion transformation to a looped schedule. It states that given a valid looped schedule, two adjacent loops having the same iteration count can be fused to yield another valid schedule if the sets of actors invoked by the fused loops are mutually disjoint.

Lemma 4.2: Suppose that S is a valid schedule for an SDF graph G, and suppose that S contains a subschedule $S_0 = L_1 L_2$, where $L_1 = (nB_1)$ and $L_2 = (nB_2)$ are two schedule loops having identical iteration counts and arbitrary bodies B_1 and B_2, respectively. Assume also that $actors(B_1) \cap actors(B_2) = \varnothing$. Then replacing S_0 with the schedule loop $S_0' = (nB_1 B_2)$ in S results in a valid schedule for G.

As a counter-example that illustrates the need for the assumption that $actors(B_1) \cap actors(B_2) = \varnothing$, consider the SDF graph in Figure 4.10. One can easily verify that the looped schedule $A(2B)(2CCBB)CC$ is a valid schedule for this SDF graph. Observe that although the two schedule loops in this schedule have a common iteration count, they both contain instances of the actor B, and thus these loops do not satisfy the hypotheses of Lemma 4.2. If we fuse these two loops, we obtain the schedule $A(2BCCBB)CC$. The invocation sequence generated by this new schedule, $ABCCBBBCCBBCC$, terminates on the edge $B \to C$ at invocation of C_2. Thus, the fusion of the schedule loops $(2B)$ and $(2CCBB)$ converts a valid schedule into a schedule that is not valid.

Proof of Lemma 4.2: Let S' denote the schedule that results from replacing S_0 with S_0' in S. By construction of S', we have that for all actors A in G, $inv(A, S') = inv(A, S)$. Since S is valid, and hence periodic, it follows that S' is also periodic. It remains to be shown that S' is admissible.

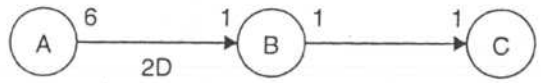


Figure 4.10. An SDF graph used to illustrate that the fusion of two adjacent schedule loops in a valid looped schedule is not always a legitimate transformation.

Clearly S' is admissible if for each edge α in G, S' does not terminate on α. There are four cases to consider here:

1) $(src(\alpha), snk(\alpha) \in actors(B_1))$ **or** $(src(\alpha), snk(\alpha) \in actors(B_2))$

2) $(src(\alpha) \notin (actors(B_1) \cup actors(B_2)))$ **or** $(snk(\alpha) \notin (actors(B_1) \cup actors(B_2)))$

3) $(src(\alpha) \in actors(B_1))$ **and** $(snk(\alpha) \in actors(B_2))$

4) $(src(\alpha) \in actors(B_2))$ **and** $(snk(\alpha) \in actors(B_1))$

Case 1:

$(src(\alpha), snk(\alpha) \in actors(B_1))$ **or** $(src(\alpha), snk(\alpha) \in actors(B_2))$.

Let i be that member of $\{1, 2\}$ such that $src(\alpha), snk(\alpha) \in actors(B_i)$, and observe that since $actors(B_1) \cap actors(B_2) = \varnothing$, $projection(S_0, actors(B_i)) = (nB_i) = projection(S_0', actors(B_i))$, and thus

$$projection(S', actors(B_i)) = projection(S, actors(B_i)). \tag{4.3}$$

Now, (S' terminates on α)

$\Rightarrow$($projection(S', actors(B_i))$ terminates on α) (by Fact 4.4b)

$\Rightarrow$($projection(S, actors(B_i))$ terminates on α) (by (4.3))

$\Rightarrow$(S is not valid) (by Fact 4.4a).

By contraposition, it follows that under the assumptions of Lemma 4.2, S' does not terminate on α.

Case 2: $(src(\alpha) \notin (actors(B_1) \cup actors(B_2)))$ **or** $(snk(\alpha) \notin (actors(B_1) \cup actors(B_2)))$

By Corollary 4.1(b), S' does not terminate on α.

Case 3: $(src(\alpha) \in actors(B_1))$ **and** $(snk(\alpha) \in actors(B_2))$.

Let r be any positive integer such that $1 \le r \le inv(S_0, S)$; let t_r denote the number of tokens on α just prior to the rth invocation of S_0 in S; and observe that since S is admissible, we must have

$$t_r + n(inv(src(\alpha), B_1))prd(\alpha) \ge n(inv(snk(\alpha), B_2)cns(\alpha)). \tag{4.4}$$

Now, by construction of S', we have that

$$\text{for all actors } A \in actors(G),\ inv(A, S_0') = inv(A, S_0). \tag{4.5}$$

Thus, the number of tokens on α just prior to the rth invocation of S_0' in S' is equal to t_r, and S' does not terminate on α during the rth invocation of S_0 if there is a sufficient number of tokens on α prior to each of the n invocations of B_2 — that is, if

$\forall(k \in \{1, 2, \ldots, n\})$,

$t_r + k\,inv(src(\alpha), B_1)prd(\alpha) - (k-1)\,inv(snk(\alpha), B_2)cns(\alpha) \ge inv(snk(\alpha), B_2)cns(\alpha)$

,

which is equivalent to

$$\forall(k \in \{1, 2, \ldots, n\}), t_r + k(inv(src(\alpha), B_1))prd(\alpha) \ge k\,inv(snk(\alpha), B_2)cns(\alpha) \tag{4.6}$$

By Fact 4.9, (4.6) is guaranteed by (4.4), and since (4.4) holds for all invocation numbers r, it follows that S' does not terminate on α during an invocation of S_0'. Furthermore, from Corollary 4.1(a), S' cannot terminate at an invocation that is not part of S_0'. We conclude that S' does not terminate on α.

Case 4: $(src(\alpha) \in actors(B_2))$ **and** $(snk(\alpha) \in actors(B_1))$.

Again, let r be any positive integer such that $1 \le r \le inv(S_0, S)$, and let t_r denote the number of tokens on α just prior to the rth invocation of S_0 in S. Then since S is admissible,

$$t_r \ge n(inv(snk(\alpha), B_1))cns(\alpha). \tag{4.7}$$

Now clearly, S' does not terminate on α during the rth invocation of S_0' if

$\forall(k \in \{1, 2, \ldots, n\})$,

$t_r + (k-1)(inv(src(\alpha), B_2))prd(\alpha) - (k-1)\,inv(snk(\alpha), B_1)cns(\alpha)$

$\ge inv(snk(\alpha), B_1)cns(\alpha)$,

which is equivalent to

$$\forall(k \in \{1, 2, \ldots, n\}), t_r + (k-1)inv(src(\alpha), B_2)prd(\alpha)$$
$$t_r + (k-1)inv(src(\alpha), B_2)prd(\alpha) \ge k\,inv(snk(\alpha), B_1)cns(\alpha). \tag{4.8}$$

Now, it is easily seen that (4.7) implies (4.8). Since this analysis holds for any choice of r, S' does not terminate on α during an invocation of S_0', and from Corollary 4.1(a), S' cannot terminate at an invocation that is not part of S_0', so we conclude that S' does not terminate on α.

Our treatment of cases 1-4 shows that for any edge α contained in G, S' does not terminate on α. ■

The following theorem establishes a sufficient condition for valid application of the factoring transformation. The condition is that the sets of actors invoked by the factored loops are all mutually disjoint. Clearly, this condition is always satisfied when working with single appearance schedules, and thus a major consequence of Theorem 4.1 is that factoring cannot convert a valid single appearance schedule into a schedule that is not valid.

Theorem 4.1: Suppose that S is a valid schedule for an SDF graph G, and suppose that $L = (m(n_1S_1)(n_2S_2)\ldots(n_kS_k))$ is a schedule loop in S of any nesting depth such that $(1 \le i < j \le k) \Rightarrow actors(S_i) \cap actors(S_j) = \varnothing$. Suppose also that γ is any positive integer that divides $n_1, n_2, \ldots, n_k$; let L' denote the schedule loop $(\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_kS_k))$; and let S' denote the schedule that results from replacing L with L' in S. Then S' is a valid schedule for G.

Proof: We will prove this theorem by induction on k.

First, let S' denote the schedule that results from replacing L with L' in S, and observe that for $k = 1$, L and L' generate the same invocation sequence, and thus S and S' generate the same invocation sequence. We conclude that S' is valid for $k = 1$, and thus Theorem 4.1 holds for $k = 1$.

Second, consider the case $k = 2$. Then $L = (m(n_1S_1)(n_2S_2))$ and $L' = (\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2))$. Now, observe that L generates the same invocation sequence as the schedule loop $\hat{L} = (m(\gamma(\gamma^{-1}n_1S_1))(\gamma(\gamma^{-1}n_2S_2)))$, so replacing L with $\hat{L}$ in the valid schedule S yields another valid schedule $\hat{S}$. Since, by assumption $actors(S_1) \cap actors(S_2) = \varnothing$, Lemma 4.2 guarantees that replacing $\hat{L}$ with $\hat{L}' = (m(\gamma(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)))$ in $\hat{S}$ yields a third valid schedule $\hat{S}'$. But, clearly $\hat{L}'$ and L' generate the same invocation sequence, so replacing $\hat{L}'$ with L' in $\hat{S}'$ results in a valid schedule $\hat{S}''$. But by construction, $\hat{S}'' = S'$, and thus S' is a valid schedule for G. We conclude that Theorem 4.1 holds for $k = 2$.

Now suppose that Theorem 4.1 holds whenever $k \le k'$, for some $k' \ge 2$. We will show that this implies the validity of Theorem 4.1 for $k \le (k'+1)$. For $k = k'+1$,

$$L = (m(n_1S_1)(n_2S_2)\ldots(n_{k'+1}S_{k'+1})), \text{ and}$$

$$L' = (\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'+1}S_{k'+1})).$$

Let S_a denote the schedule that results from replacing L with the schedule loop $L_a = (m(1(n_1S_1)(n_2S_2)\ldots(n_{k'}S_{k'}))(n_{k'+1}S_{k'+1}))$ in S. Since L_a and L generate the same invocation sequence, S_a generates the same invocation sequence as S. Now Theorem 4.1 for $k = k'$ guarantees that replacing $(1(n_1S_1)(n_2S_2)\ldots(n_{k'}S_{k'}))$ with $(\gamma(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'}S_{k'}))$ in S_a results in a valid schedule S_b.

Observe that S_b is the schedule S with L replaced by

$L_b = (m(\gamma(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'}S_{k'}))(n_{k'+1}S_{k'+1}))$.

Theorem 4.1 for $k = 2$ guarantees that replacing L_b with

$$L_c = (\gamma m(1(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_{k'}S_{k'}))(\gamma^{-1}n_{k'+1}S_{k'+1}))$$

yields another valid schedule S_c. Now clearly L_c generates the same invocation sequence as L', so replacing L_c with L' in S_c yields a valid schedule S_d. But, by construction, $S_d = S'$, so S' is a valid schedule for G.

We have shown that Theorem 4.1 holds for $k = 1$ and $k = 2$, and we have shown that if the result holds for $k \le k'$, then it holds $k \le (k'+1)$. We conclude that Theorem 4.1 holds for all k. ■

We have demonstrated that factoring may decrease the buffer memory requirement for a schedule. Although the transformation is not guaranteed to always decrease the buffer memory requirement, factoring never increases the buffer memory requirement. This is established by the following theorem.

Theorem 4.2: As in Theorem 4.1, assume that S is a valid schedule for an SDF graph G; $L = (m(n_1S_1)(n_2S_2)\ldots(n_kS_k))$ is a schedule loop in S of any nesting depth such that $(1 \le i < j \le k) \Rightarrow actors(S_i) \cap actors(S_j) = \varnothing$; and γ is a positive integer that divides $n_1, n_2, \ldots, n_k$. Let L' denote the schedule loop $(\gamma m(\gamma^{-1}n_1S_1)(\gamma^{-1}n_2S_2)\ldots(\gamma^{-1}n_kS_k))$, and let S' denote the schedule that results from replacing L with L' in S. Then $max_tokens(e, S') \le max_tokens(e, S)\ \forall e$.

Proof: Let α be an arbitrary edge in G. We consider three cases.

Case 1: $(src(\alpha) \notin actors(L))$ **or** $(snk(\alpha) \notin actors(L))$. From Lemma 4.1(b), we have that

$$\forall(i \in \{1, 2, \ldots, inv(snk(\alpha), S)\}), P(\alpha, i, S') = P(\alpha, i, S),$$

and from Fact 4.3(a), it follows that

$$\forall(i \in \{1, 2, \ldots, inv(snk(\alpha), S)\}), T(\alpha, i, S') = T(\alpha, i, S).$$

Thus, from Fact 4.3(c), we have $max_tokens(\alpha, S') = max_tokens(\alpha, S)$.

Case 2: For some $j \in \{1, 2, \ldots, k\}$, $src(\alpha), snk(\alpha) \in actors(S_j)$. Then since $actors(S_1) \cap actors(S_2) \cap \ldots \cap actors(S_k) = \varnothing$, it is easily verified from the construction of S', that $projection(S, \{src(\alpha), snk(\alpha)\})$ generates the same invocation sequence as $projection(S', \{src(\alpha), snk(\alpha)\})$. From Fact 4.3(b) and Fact 4.3(a), $T(\alpha, i, S') = T(\alpha, i, S)$, and thus

$$max_tokens(\alpha, S') = max_tokens(\alpha, S).$$

Case 3: $(src(\alpha) \in actors(S_p))$ **and** $(snk(\alpha) \in actors(S_q))$, where $p, q \in \{1, 2, \ldots, k\}$ and $p \neq q$. We define

$$r_1 \equiv max(\{T(\alpha, i, S) | snk(\alpha)_i \text{ is part of } L\});$$

$$r_1' \equiv max(\{T(\alpha, i, S') | snk(\alpha)_i \text{ is part of } L'\});$$

$$r_2 \equiv max(\{T(\alpha, i, S) | snk(\alpha)_i \text{ is not part of } L\});$$

$$r_2' \equiv max(\{T(\alpha, i, S') | snk(\alpha)_i \text{ is not part of } L'\}).$$

Then clearly,

$$max_tokens(\alpha, S) = max(\{r_1, r_2\}) \text{ and}$$
$$max_tokens(\alpha, S') = max(\{r_1', r_2'\}). \tag{4.9}$$

Now, if in S, $snk(\alpha)_i$ is not part of L, then clearly by the construction of L', $snk(\alpha)_i$ is not part of L' in S', and from Lemma 4.1(a) and Fact 4.3(a), we have that $T(\alpha, i, S') = T(\alpha, i, S)$, and thus

$$r_2 = r_2'. \tag{4.10}$$

On the other hand, if $snk(\alpha)_i$ is part of L in S, and thus $snk(\alpha)_i$ is part of L' in S', we define

$$\Delta \equiv (n_p \times inv(src(\alpha), S_p) \times prd(\alpha)) - \quad ;$$
$$(n_q \times inv(snk(\alpha), S_q) \times cns(\alpha))$$

$M(\alpha, j) \equiv max(\{T(\alpha, i, S) | snk(\alpha)_i \text{ is part of the } j\text{th invocation of } L\})$; and
$M'(\alpha, j) \equiv max(\{T(\alpha, i, S') | snk(\alpha)_i \text{ is part of the } j\text{th invocation of } L'\})$.
Also, we define x_j to denote the number of tokens on α just before the jth invocation of L (L') in S (S'), and we define y_j to denote the number of tokens on α just after the jth invocation of L (L') in S (S').

Clearly, if $\Delta \geq 0$, then during a particular invocation of L in an execution of S, the maximum number of tokens on α is attained just after the last invocation of $(n_p S_p)$. Similarly in an execution of S', the maximum number of tokens on α during the jth invocation of L' is attained just after the last invocation of $(\gamma^{-1} n_p S_p)$. Thus, if $p < q$, then

$$M(\alpha, j) = x_j + m n_p inv(src(\alpha), S_p) prd(\alpha) - (m-1) n_q inv(snk(\alpha), S_q) cns(\alpha)$$
$$= x_j + m\Delta + n_q inv(snk(\alpha), S_q) cns(\alpha),$$

and similarly,

$$M'(\alpha, j) = x_j + (\gamma m) \frac{n_p}{\gamma} inv(src(\alpha), S_p) prd(\alpha) - (\gamma m - 1) \frac{n_q}{\gamma} inv(snk(\alpha), S_q) cns(\alpha)$$
$$= x_j + m\Delta + \frac{n_q}{\gamma} inv(snk(\alpha), S_q) cns(\alpha).$$

Thus, since $\gamma \geq 1$, we have that $M'(\alpha, j) \leq M(\alpha, j)$, and since this holds for all j, r_1' cannot exceed r_1. From (4.84.9) and (4.10), it follows that $max_tokens(\alpha, S') \leq max_tokens(\alpha, S)$.

If $(\Delta \geq 0)$ and $(p > q)$, then clearly y_j cannot be less than $M(\alpha, j)$ nor $M'(\alpha, j)$ for any j. Thus,

$$max_tokens(\alpha, S) = max(\{r_2, y_1, y_2, \ldots, y_{inv(L, S)}\}) = max_tokens(\alpha, S').$$

If $(\Delta < 0)$ and $(p < q)$, then for any j,

$$M(\alpha, j) = x_j + n_p inv(src(\alpha), S_p) prd(\alpha), \text{ and}$$

$$M'(\alpha, j) = x_j + \frac{n_p}{\gamma} inv(src(\alpha), S_p) prd(\alpha).$$

Thus, $M'(\alpha, j) \leq M(\alpha, j)$ for all j, and we have $r_1' \leq r_1$. From (4.84.9) and (4.10), we conclude that $max_tokens(\alpha, S') \leq max_tokens(\alpha, S)$.

Finally, if $(\Delta < 0)$ and $(p > q)$, then clearly $M(\alpha, j) = M'(\alpha, j) = x_j$ for all j. Thus $r_1' = r_1$, and (4.84.9) and (4.10) yield that

$$max_tokens(\alpha, S') = max_tokens(\alpha, S).$$

Any edge α in G must fall into the domain of case 1, case 2 or case 3, and in each of these cases, we have established that

$$max_tokens(\alpha, S') \le max_tokens(\alpha, S). \blacksquare$$

Recall that the definition of *buffer memory requirement* assumes that each buffer is implemented as a separate, contiguous block of storage, and thus Theorem 4.2 does not necessarily apply under more flexible buffer implementations — such as when storage is shared between multiple buffers that are active (contain unread data) in mutually disjoint segments of time. In [Bhat94a], shared buffers and buffers that do not necessarily reside in contiguous memory locations are discussed.

4.5 Reduced Single Appearance Schedules

Definition 4.4: If Λ is either a schedule loop or a looped schedule, we say that Λ is **coprime** if not all iterands of Λ are schedule loops, or if all iterands of Λ are schedule loops, and there does not exist an integer $j > 1$ that divides all of the iteration counts of the iterands of Λ.

For example, the schedule loops $(3(4A)(2B))$ and $(10(7C))$ are both non-coprime, while the loops $(5(3A)(7B))$ and $(70C)$ are coprime. Similarly, the looped schedules $(4AB)$ and $(6AB)(3C)$ are both non-coprime, while the schedules $A(7B)(7C)$ and $(2A)(3B)$ are coprime. From the discussion in the previous section, we know that non-coprime schedules or loops may result in much higher buffer memory requirements than their factored counterparts.

Definition 4.5: Given a single appearance schedule S, we say that S is **fully reduced** if S is coprime and every schedule loop contained in S is coprime.

In this section, we show that we can always convert a valid single appearance schedule that is not fully reduced into a valid fully reduced schedule, and thus, we can always avoid the potential overhead associated with using non-coprime schedule loops over their corresponding factored forms. First, however, we show that any fully reduced schedule has unit blocking factor. This implies that any schedule that has blocking factor greater than one is not fully reduced. Thus, if we decide to implement a schedule that has nonunity blocking factor, then we risk introducing a higher buffer memory requirement.

Theorem 4.3: Suppose that S is a valid, fully reduced schedule, and $L = (i_L B_L)$ is a schedule loop in S. Then for each $A \in actors(B_L)$,

$$inv(A, B_L) = \frac{\mathbf{q}(A)}{gcd(\{\mathbf{q}(A') | (A' \in actors(B_L))\})}.$$

Proof: Observe that for each $A \in actors(B_L)$, we have $J(S)\mathbf{q}(A) = k \times inv(A, B_L)$, where $k = inv(B_L, S)$. Thus, it suffices to show that $gcd(\{inv(A', B_L) | A' \in actors(B_L)\}) = 1$.

First suppose that not all iterands of L are schedule loops. Then $inv(A, B_L) = 1$ for some $A \in actors(B_L)$, and we are done.

Now suppose that all iterands of L are schedule loops, and suppose that j is an arbitrary integer that is greater than one. Then, since S is fully reduced, j does not divide at least one of the iteration counts associated with the iterands of L. Define $i_0 = 1$ and let L_1 denote one of the iterands of L whose iteration count i_1 is not divisible by $j = j/(gcd(\{j, i_0\}))$. Again, since S is fully reduced, if all iterands of L_1 are schedule loops, then there exists an iterand L_2 of L_1 such that $j = j/(gcd(\{j, i_0 i_1\}))$ does not divide the iteration count i_2 of L_2. Similarly, if all iterands of L_2 are schedule loops, there exists an iterand L_3 of L_2 whose iteration count i_3 is not divisible by $j = j/(gcd(\{j, i_0 i_1 i_2\}))$.

Continuing in this manner, we get a sequence $(L_1, L_2, L_3 \ldots)$ such that the iteration count i_k of each L_k is not divisible by $j = j/(gcd(\{j, i_0 i_1 \ldots i_{k-1}\}))$. Since S contains a finite number of loops, we cannot continue this process indefinitely — for some $m \ge 1$, not all iterands of L_m are schedule loops. Thus, there is an actor A that is an iterand of L_m. Since S is a single appearance schedule,

$$inv(A, B_L) = inv(L_2, L_1) inv(L_3, L_2) \ldots inv(L_{m-1}, L_m) inv(A, L_m) = i_0 i_1 \ldots i_m \quad (4.11)$$

By the selection of the L_k's, $j/(gcd(\{j, i_0 i_1 \ldots i_{m-1}\}))$ does not divide i_m, and thus, from (4.11), j does not divide $inv(A, B_L)$.

We have shown that given any integer $j > 1$, there exists an $A \in actors(B_L)$ such that $inv(A, B_L)$ is not divisible by j. It follows that $gcd(\{inv(A', B_L) | A' \in actors(B_L)\}) = 1$. $\blacksquare$

Corollary 4.2: Suppose that G is a connected SDF graph and S is a valid fully reduced single appearance schedule for G. Then $J(S) = 1$.

Proof: Since S is fully reduced, so is $(1S)$. Applying Theorem 4.3 with $i_L = 1$ and $B_L = S$, we have that

$$inv(A, S) = \frac{\mathbf{q}(A)}{gcd(\{\mathbf{q}(A') | (A' \in actors(S))\})}. \quad (4.12)$$

Since $actors(S) = actors(G)$, we have from Fact 3.4 that the denominator in (4.12) is unity. Thus S invokes each actor A exactly $\mathbf{q}(A)$ times. $\blacksquare$

Theorem 4.4: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, and S is a single appearance schedule for G. Then there exists a valid, fully reduced schedule S' such that $lexorder(S') = lexorder(S)$, and $max_tokens(e, S') \le max_tokens(e, S)$, for each $e \in E$.

Proof: We prove this theorem by construction. This construction process can easily be automated to yield an efficient algorithm for synthesizing a valid fully reduced schedule from an arbitrary valid single appearance schedule.

Given a looped schedule Ψ, we denote the set of schedule loops in Ψ that are not coprime by $non\text{-}coprime(\Psi)$. Now suppose that S is a valid single appearance schedule for G, and let $\lambda_1 = (m(n_1\Psi_1)(n_2\Psi_2)\ldots(n_k\Psi_k))$ be any innermost member of $non\text{-}coprime(S)$ — that is, λ_1 is non-coprime, but every schedule loop nested within λ_1 is coprime. From Theorems 4.1 and 4.2, replacing λ_1 with

$$\lambda_1' = (\gamma m(\gamma^{-1}n_1\Psi_1)(\gamma^{-1}n_2\Psi_2)\ldots(\gamma^{-1}n_k\Psi_k)),$$

where $\gamma = gcd(\{n_1, n_2, \ldots, n_k\})$, yields another valid single appearance schedule S_1 such that $max_tokens(e, S_1) \le max_tokens(e, S)$, for all $e \in E$. Furthermore, λ_1' is coprime, and since every schedule loop nested within λ_1 is coprime, every loop nested within λ_1' is coprime as well. Now let λ_2 be any innermost member of $non\text{-}coprime(S_1)$, and observe that λ_2 cannot equal λ_1'. Theorems 4.1 and 4.2 guarantee a replacement λ_2' for λ_2 in S_1 that leads to another valid single appearance schedule S_2 such that $max_tokens(e, S_2) \le max_tokens(e, S)$, for all $e \in E$. If we continue this process, it is clear that no replacement loop λ_k' ever replaces one of the previous replacement loops $\lambda_1', \lambda_2', \ldots, \lambda_{k-1}'$, since these loops and the loops nested within these loops are already coprime. Also, no replacement changes the total number of schedule loops in the schedule. It follows that we can continue this process only a finite number of times — eventually, we will arrive at an S_n such that $non\text{-}coprime(S_n)$ is empty.

Now if S_n is a coprime looped schedule, we are done. Otherwise, S_n is of the form $(p_1T_1)(p_2T_2)\ldots(p_mT_m)$, where $\gamma' \equiv gcd(\{p_1, p_2, \ldots, p_m\}) > 1$. Applying Theorem 4.1 to the schedule $(1S_n) = (1(p_1T_1)(p_2T_2)\ldots(p_mT_m))$, we have that

$$(\gamma'((\gamma')^{-1}p_1T_1)((\gamma')^{-1}p_2T_2)\ldots((\gamma')^{-1}p_mT_m))$$

is a valid schedule for G. From the definition of a valid schedule, it follows that

$$S_n' \equiv ((\gamma')^{-1}p_1T_1)((\gamma')^{-1}p_2T_2)\ldots((\gamma')^{-1}p_mT_m)$$

is also a valid schedule, and by construction of S_n and S_n', S_n' is a coprime single appearance schedule, and all schedule loops in S_n' are coprime. Thus, S_n' is a valid fully reduced schedule for G. Furthermore, since $(1S_n)$ generates the same invocation sequence as S_n clearly $max_tokens(e, (1S_n)) = max_tokens(e, S_n)$ for all $e \in E$. From Theorem 4.2, $max_tokens(e, S_n') \le max_tokens(e, (1S_n))$ for all $e \in E$, and thus $max_tokens(e, S_n') \le max_tokens(e, S)$ for all $e \in E$.

It is easily verified that none of the transformations in the derivation of S_n' affect the lexical ordering, and thus $lexorder(S_n') = lexorder(S)$. ■

4.6 Subindependence

Since valid single appearance schedules implement the full repetition inherent in an SDF graph without requiring subroutines or code duplication, we examine the topological conditions required for such schedules to exist. First, suppose that G is a connected, consistent acyclic SDF graph containing n actors. Then we can take some root actor R_1 of G and fire all $\mathbf{q}_G(R_1)$ invocations of R_1 in succession. After all invocations of R_1 have fired, we can remove R_1 from G, pick a root actor R_2 of the new acyclic SDF graph, and schedule its $\mathbf{q}_G(R_2)$ repetitions in succession. Clearly, we can repeat this process until no actors are left, to obtain the single appearance schedule $(\mathbf{q}_G(R_1)R_1)(\mathbf{q}_G(R_2)R_2)\ldots(\mathbf{q}_G(R_n)R_n)$ (schedules of this form are called **flat single appearance schedules**) for G. Thus, we see that any consistent acyclic SDF graph has a valid single appearance schedule.

Also, observe that if G is an arbitrary connected, consistent SDF graph, then we can cluster the subgraph associated with each nontrivial strongly connected component of G. Clustering a strongly connected component into a single actor Ω never results in deadlock since there can be no cycle containing Ω. Since clustering all strongly connected components yields an acyclic graph, it follows from Fact 4.4 and Fact 4.6 that *G has a valid single appearance schedule if and only if each strongly connected component has a valid single appearance schedule.*

Observe that we must, in general, analyze a strongly connected component subgraph Θ as a separate entity since G may have a valid single appearance schedule even if there is an actor A in Θ for which we cannot fire all $\mathbf{q}_G(A)$ invocations in succession. The key is that $\mathbf{q}_\Theta(A)$ may be less than $\mathbf{q}_G(A)$, so we

may be able to generate a single appearance subschedule for Θ; for example, we may be able to schedule A $\mathbf{q}_\Theta(A)$ times in succession. Since we can schedule G so that the subschedule for Θ appears only once, this will translate into a single appearance schedule for G. For example, in Figure 4.11(a), it can be verified that $\mathbf{q}(A, B, C) = (10, 4, 5)^T$, but we cannot fire so many invocations of A or B in succession. However, consider the strongly connected component subgraph $\Theta' \equiv subgraph(\{A, B\})$. Then we obtain $\mathbf{q}_{\Theta'}(A) = 5$ and $\mathbf{q}_{\Theta'}(B) = 2$, and we immediately see that $\mathbf{q}_{\Theta'}(B)$ invocations of B can be fired in succession to yield a subschedule for Θ'. The SDF graph that results from clustering Θ' is shown in Figure 4.11(b). This leads to the valid single appearance schedule $(2(2B)(5A))(5C)$.

Theorem 4.5: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, S is a single appearance schedule (not necessarily fully reduced) for G, and k is any positive integer. Then there exists a valid single appearance schedule S' such that $J(S') = k$ (i.e, the blocking factor is k), $lexorder(S') = lexorder(S)$, and $max_tokens(e, S') \le max_tokens(e, S)$, for each $e \in E$.

Proof: From Theorem 4.4, there is a valid, fully reduced schedule S'' such that $lexorder(S'') = lexorder(S)$, and $max_tokens(e, S'') \le max_tokens(e, S)$, for each $e \in E$. Clearly, since S'' is fully reduced, $(1S'')$ is also fully reduced. Thus, applying Fact 4.3 with $L = (1S'')$, we have that $J(S'') = 1$. Thus (kS'') is a valid schedule that has blocking factor k, and has the same lexical ordering as S.

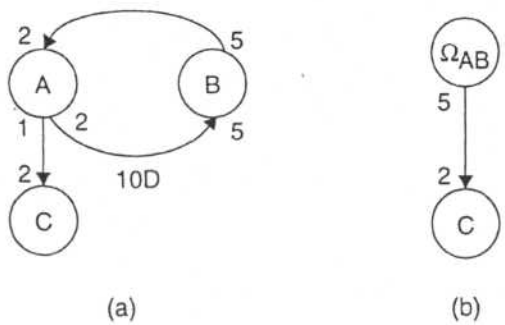


Figure 4.11. An example of how clustering strongly connected components can aid in generating compact looped schedules.

Furthermore, since S'' is a valid schedule, clearly $max_tokens(e, (kS'')) = max_tokens(e, S'')$ for all $e \in E$, and thus, $max_tokens(e, (kS'')) \le max_tokens(e, S)$ for all $e \in E$. ■

Corollary 4.3: Suppose that G is an SDF graph that has a valid single appearance schedule (G need not be connected). Then G has a valid single appearance schedule for all blocking vectors.

Proof: Suppose that S is a valid single appearance schedule for G, let $\kappa_1, \kappa_2, \ldots, \kappa_n$ denote the connected components of G, let $\mathbf{J}'(\kappa_1, \kappa_2, \ldots, \kappa_n) \equiv (z_1, z_2, \ldots, z_n)$ be an arbitrary blocking vector for G, and for $1 \le i \le n$, let S_i denote the projection of S onto κ_i. Then from Fact 4.4, each S_i is a valid single appearance schedule for the corresponding $subgraph(\kappa_i)$. From Theorem 4.5, for $1 \le i \le n$, there exists a valid single appearance schedule S_i' of blocking factor z_i for $subgraph(\kappa_i, G)$. Since the κ_i's are mutually disjoint and non-adjacent, it follows that $S_1'S_2'\ldots S_n'$ is a valid single appearance schedule of blocking vector $\mathbf{J}'$ for G. ■

The condition for the existence of a valid single appearance schedule can be expressed in terms of a form of precedence independence, which is specified in the following definition.

Definition 4.6: Suppose that G is a connected, sample rate consistent SDF graph. If Z_1 and Z_2 are disjoint nonempty subsets of $actors(G)$, we say that Z_1 *is* **subindependent** *of* Z_2 *in* G if for every edge α in G such that $src(\alpha) \in Z_2$ and $snk(\alpha) \in Z_1$, we have $del(\alpha) \ge TNSE(\alpha, G)$. We occasionally drop the "in G" qualification if G is understood from context. Also, if (Z_1 is subindependent of Z_2) **and** ($Z_1 \cup Z_2 = actors(G)$), then we say that Z_1 *is subindependent in* G, and we say that Z_1 and Z_2 form a **subindependent partition** of G.

In other words, Z_1 is subindependent of Z_2 if given a minimal periodic schedule for G, data produced by Z_2 is never consumed by Z_1 in the same schedule period in which it is produced. Thus, at the beginning of each schedule period, all of the data required by Z_1 from Z_2 for that schedule period is available at the inputs of Z_1. For example, let G denote the SDF graph in Figure 4.12. Here $\mathbf{q}(A, B, C, D) = (2, 1, 2, 2)^T$, and we see that $\{A\}$ is subindependent of $\{C\}$; $\{A, D\}$ and $\{B, C\}$ form a subindependent partition of G; and trivially, $\{A, B, C\}$ is subindependent of $\{D\}$.

The following properties of subindependence follow immediately from Definition 4.6.

Fact 4.10: Suppose that G is a connected, sample rate consistent SDF graph, and X, Y and Z are disjoint, nonempty subsets of $actors(G)$. Then

(a) (X is subindependent of Z) **and** (Y is subindependent of Z) $\Rightarrow$ ($X \cup Y$) is subindependent of Z

(b) (X is subindependent of Y) **and** (X is subindependent of Z) $\Rightarrow X$ is subindependent of ($Y \cup Z$)

Recall that an arbitrary consistent SDF graph has a valid single appearance schedule if and only if each strongly connected component has a single appearance schedule. The following theorem gives necessary and sufficient conditions for a strongly connected SDF graph to have a valid single appearance schedule.

Theorem 4.6: Suppose that G is a nontrivial, consistent, strongly connected SDF graph. Then G has a valid single appearance schedule if and only if there exists a nonempty proper subset $X \subset actors(G)$ such that

(1) X is subindependent of $(actors(G) - X)$ in G; and

(2) $subgraph(X, G)$ and $subgraph(actors(G) - X, G)$ both have valid single appearance schedules.

Proof: ($\Leftarrow$ direction). Let S and T denote valid single appearance schedules for $Y \equiv subgraph(X, G)$ and $Z \equiv subgraph((actors(G) - X), G)$, respectively; let $y_1, y_2, \ldots, y_k$ denote the connected components of Y; and let $z_1, z_2, \ldots, z_l$ denote the connected components of Z. From Corollary 4.3, we can assume without loss

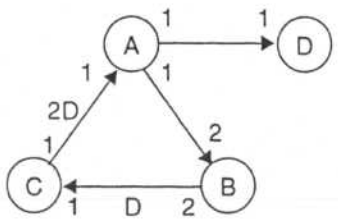


Figure 4.12. An example used to illustrate the concept of subindependence.

of generality that for $1 \le i \le k$, $\mathbf{J}_S(y_i) = q_G(y_i)$, and that for $1 \le i \le l$, $\mathbf{J}_T(z_i) = q_G(z_i)$. From Fact 4.5, it follows that S invokes each $A \in X$ $\mathbf{q}_G(A)$ times, and T invokes each $A \in (actors(G) - X)$ $\mathbf{q}_G(A)$ times, and since X is subindependent in G, it follows that ST, the schedule obtained by appending T to S, is a valid single appearance schedule (of blocking factor one) for G.

($\Rightarrow$ direction). Suppose that S is a valid single appearance schedule for G. From Theorem 4.5, we can assume without loss of generality that S has blocking factor one, and from Fact 4.1, there exists a valid single appearance schedule S' that has blocking factor one and contains no one-iteration loops. Then S' can be expressed as $S_a S_b$, where S_a and S_b are nonempty single appearance subschedules of S' that are not encompassed by a loop, since if S' is a schedule loop $(n(\ldots)(\ldots)\ldots(\ldots))$, then $gcd(\{\mathbf{q}_G(A) | (A \in actors(G))\}) \ge n$ so S' does not have unity blocking factor — a contradiction. Since $S_a S_b$ is a minimal, valid single appearance schedule for G, every actor $A \in actors(S_a)$ is invoked $\mathbf{q}_G(A)$ times before any actor outside of $actors(S_a)$ is invoked. It follows that $actors(S_a)$ is subindependent of $actors(S_b)$ in G. Also, by Fact 4.4, S_a is a valid single appearance schedule for $subgraph(actors(S_a))$ and S_b is a valid single appearance schedule for $subgraph(actors(S_b))$. ■

Theorem 4.6 states that in order for a strongly connected SDF graph G to have a valid single appearance schedule, it must have in addition, a subindependent partition Z_1, Z_2. We can construct a valid single appearance schedule for G by constructing a valid single appearance schedule for all invocations associated with Z_1 and then concatenating a valid single appearance schedule for all invocations associated with Z_2. By repeatedly applying this type of decomposition, we can construct single appearance schedules whenever they exist. We will elaborate on this extensively in the following chapter.

The following theorem presents a simple topological condition for the existence of a subindependent partition that leads to an efficient algorithm for finding a subindependent partition whenever one exists.

Theorem 4.7: Suppose that G is a nontrivial, strongly connected, consistent SDF graph. From G, remove all edges α for which $del(\alpha) \ge TNSE(\alpha, G)$, and call the resulting SDF graph G'. Then G has a subindependent partition if and only if G' is not strongly connected. Furthermore, if G' is not strongly connected, then any root strongly connected component Z of G' is subindependent of $(actors(G) - Z)$ in G.

Proof: First suppose that G' is not strongly connected, and let Z_1 be any root strongly connected component of G'. Thus, no edge in G that is directed from a member of $(actors(G) - Z_1)$ to a member of Z_1 is contained in G'. Thus, by the construction of G', for each edge α in G directed from a member of $(actors(G) - Z_1)$ to a member of Z_1, we have $del(\alpha) \geq TNSE(\alpha, G)$. It follows that Z_1 is subindependent in G. Thus, since Z_1 is an arbitrary root strongly connected component of G', we have shown that if G' is not strongly connected, then G has a subindependent partition and any root strongly connected component of G' is subindependent in G.

To complete the proof, we show that whenever G has a subindependent partition, G' is not strongly connected. If G has a subindependent partition, then $actors(G)$ can be partitioned into Z_1 and Z_2 such that Z_1 is subindependent of Z_2 in G. By construction of G', there are no edges in G' directed from a member of Z_2 to a member of Z_1, so G' is not strongly connected. ■

Theorem 4.7 establishes the validity of the algorithm shown in Figure 4.13, which takes as input a nontrivial, consistent, strongly connected SDF graph G, and finds a subindependent partition of G if one exists.

Let $m = max(\{|actors(G)|, |edges(G)|\})$. The algorithm presented in Subsection 3.1 computes the repetitions vector in time $O(m)$; it is obvious that the next step of algorithm *SubindependentPartition* — removing the edges with insufficient delay — can also be performed in $O(m)$ time; Tarjan's algorithm allows the determination of the strongly connected components in $O(m)$ time [Tarj72]; and the checks in the if-else segment are clearly $O(m)$ as well. Thus, the time complexity of algorithm *SubindependentPartition* is linear in the number of actors and edges in G.

The operation of algorithm *SubindependentPartition* is illustrated in Figure 4.14. For the strongly connected SDF graph on the left side of this figure, which we denote by G, $\mathbf{q}(A, B, C, D) = (1, 10, 2, 20)^T$. Thus, the delay on the edge directed from D to B (25) exceeds the total number of tokens consumed by B in a minimal schedule period of G (20). We remove this edge to obtain the new graph depicted on the right side of Figure 4.14. Since this new SDF graph is not strongly connected, a subindependent partition of G exists: the root strongly connected component $\{A, B\}$ is subindependent of the remaining actors $\{C, D\}$ in G.


```
procedure SubindependentPartition(G)
    Compute the repetitions vector q of G.
    From G, remove each edge α for which del(α) ≥ TNSE(α, G).
    Denote the resulting graph by G'.
    Determine the strongly connected components of G'.
    if G' consists of only one strongly connected component,
    actors(G'),
        G' does not have a subindependent partition
    else
        for each strongly connected component Z
            if no member of Z has an input edge α such that
            src(α) ∉ Z
                Z is subindependent in G.
            end if
        end for
    end if
```

Figure 4.13. Algorithm for computing a subindependent partition.

4.7 Computation Graphs

The algorithms presented in this chapter can be extended in a straightforward way to the Karp-Miller computation graphs in certain cases. Recall that the computation graphs of Karp and Miller [Karp66] are like SDF graphs except that each input edge to an actor has an additional parameter called the threshold. The actor can only fire if the number of tokens on each of its input edges exceeds the

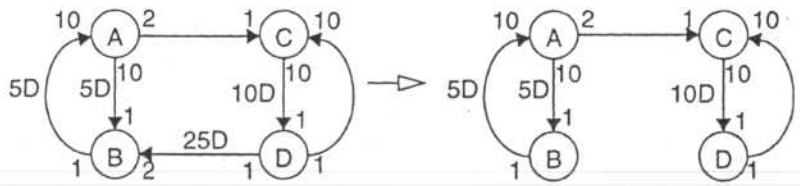


Figure 4.14. An illustration of algorithm *SubindependentPartition*.

threshold for that edge. Denoting the threshold for an edge e by $thr(e)$, if $thr(e) = cns(e)$ for all e then the computation graph is an SDF graph. If $thr(e) > cns(e)$, then we can derive a sufficient condition for a computation graph to have a periodic schedule. Denote the number of tokens on an edge e by $b(e)$. Suppose that in the computation graph, we are able to execute actors until we reach the condition

$$b(e) \geq del(e) + thr(e) - cns(e) \text{ for all } e. \tag{4.13}$$

Now consider this computation graph as an SDF graph with $b(e) - (thr(e) - cns(e))$ delays on each edge. An SDF schedule for this graph will fire an actor only if the number of tokens on each input edge e is at least $cns(e)$. This means that in the computation graph, the number of tokens on input edge e for an actor is at least $cns(e) + (thr(e) - cns(e)) = thr(e)$. Hence, the SDF schedule is a valid schedule for the computation graph. The schedule for the computation graph that leads to the state given in (4.13) is called the **preamble**. In summary, a computation graph can be statically scheduled if we can find a preamble such that (4.13) holds and if the corresponding SDF graph has a valid schedule. The schedule will consist of the preamble plus a periodic schedule constructed by invoking an SDF scheduler. If it is not possible to execute the computation graph to a state where (4.13) holds, then it is not clear whether SDF techniques can be applied. Further research is needed to deal with such cases.

Similarly, we can define subindependence for computation graphs also. In Definition 4.6, we replace the condition $del(\alpha) \geq TNSE(\alpha, G)$ by

$$del(\alpha) \geq TNSE(\alpha, G) + thr(\alpha) - cns(\alpha) \tag{4.14}$$

Again, if we consider this computation graph as an SDF graph with

$$del(\alpha) - (thr(\alpha) - cns(\alpha))$$

delays on each edge, then any SDF schedule will be a valid computation graph schedule. Furthermore, since $del(\alpha) - (thr(\alpha) - cns(\alpha)) \geq TNSE(\alpha, G)$, this SDF graph will have the same subindependent partitions as the computation graph.

5

Loose Interdependence Algorithms

The first section of this chapter presents systematic techniques for compiling SDF graphs into implementations that require minimum code size. A graph decomposition process is defined that can be used to construct single appearance schedules whenever they exist. Based on this decomposition process, a general framework for developing scheduling algorithms is defined, and it is shown that all scheduling algorithms that are constructed through this framework construct single appearance schedules whenever they exist. It is also shown that the code size optimality of the scheduling framework extends in a restricted way to SDF graphs that do not have single appearance schedules: the framework guarantees minimum code size for all actors that are not contained in subgraphs of a certain form, called *tightly interdependent subgraphs*. Section 5.2 presents a practical example to illustrate the efficacy of the scheduling framework of section 5.1.

Section 5.3 discusses considerations that must be addressed when incorporating clustering techniques into the scheduling framework, and presents a clustering technique that can be incorporated into the framework to increase the amount of buffering that occurs through registers. A large part of Section 5.3 is devoted to establishing that this clustering technique does not violate the code size minimization properties of the scheduling framework. Finally Section 5.4 relates the scheduling framework developed in this chapter to classical vectorization techniques for procedural programs.

5.1 Loose Interdependence Algorithms

Definition 5.1: Suppose that G is a sample rate consistent, nontrivial strongly connected SDF graph. Then we say that G is **loosely interdependent** if G has a subindependent partition. We say that G is **tightly interdependent** if it is not loosely interdependent.

For example, consider the strongly connected SDF graph in Figure 5.1. Here, the repetitions vector is $\mathbf{q}(A, B, C) = (3, 2, 1)^T$, and d_1, d_2 and d_3 represent the number of delays on the associated edges. From Definition 5.1, this SDF graph is loosely interdependent if and only if $(d_1 \geq 6)$ **or** $(d_2 \geq 2)$ **or** $(d_3 \geq 3)$; equivalently the graph is tightly interdependent if and only if $(d_1 < 6)$ **and** $(d_2 < 2)$ **and** $(d_3 < 3)$.

We will use the following fact, which follows immediately from the definition of loose interdependence.

Fact 5.1: If G_1 and G_2 are two isomorphic SDF graphs and G_1 is loosely interdependent, then G_2 is loosely interdependent.

The code scheduling framework in this chapter is based on the following definition, which decomposes the scheduling process into four distinct functions and defines how algorithms for these functions can be combined to generate a class of scheduling algorithms.

Definition 5.2: Let ϑ_1 be any algorithm that takes as input a nontrivial strongly connected SDF graph G, determines whether G is loosely interdependent, and if so, finds a subindependent partition of G. Le ϑ_2 be any algorithm that finds the strongly connected components of a directed multigraph. Let ϑ_3 be any algorithm that takes an acyclic SDF graph and generates a valid single appearance schedule. Finally, let ϑ_4 be any algorithm that takes a tightly interdependent SDF

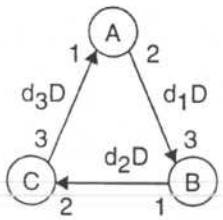


Figure 5.1. An example used to illustrate the concepts of loose and tight interdependence.

graph and generates a valid looped schedule of blocking factor one. We define the algorithm $L(\vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$ by the sequence of steps shown in Figure 5.2. This process for combining the algorithms ϑ_1, ϑ_2, ϑ_3, and ϑ_4 defines a family of algorithms $L(\bullet, \bullet, \bullet, \bullet)$, which we call **loose interdependence algorithms** because they exploit loose interdependence to decompose the input SDF graph. Given a loose interdependence algorithm $\zeta = L(\vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$, we call the component algorithms ϑ_1, ϑ_2, ϑ_3, and ϑ_4 the **subindependence partitioning algorithm** of ζ, the **strongly connected components algorithm** of ζ, the **acyclic scheduling algorithm** of ζ, and the **tight scheduling algorithm** of ζ, respectively. ■

Since nested recursive calls decompose a graph into finer and finer strongly connected components, it easy to verify that a loose interdependence algorithm always terminates on a finite input graph. Also, since the *for*-loop in step 4 replaces each Ω_i in S' with a valid looped schedule for $subgraph(Z_i)$, we know from Fact 4.4 that these replacements yield a valid looped schedule for G, and thus that the output $S_L(G)$ of a loose interdependence algorithm is always a valid schedule.

We will also make use of the following observations in the remainder of this section.

Observation 5.1: Observe that step 4 does not insert or delete appearances of actors that are not contained in a nontrivial strongly connected component Z_k. Since ϑ_3 generates a single appearance schedule for G', we have that for every actor A that is not contained in a nontrivial strongly connected component of G, $appearances(A, S_L(G)) = 1$.

Observation 5.2: If Z is a nontrivial strongly connected component of G and $A \in Z$, then since $S_L(G)$ is derived from S' by replacing the single appearance of each Ω_i, we have that

$$appearances(A, S_L(G)) = appearances(A, S_L(subgraph(Z))).$$

Observation 5.3: For each strongly connected component Z_k whose associated subgraph is loosely interdependent, L partitions Z_k into X and Y such that X is subindependent of Y in $subgraph(Z_k)$, and replaces the single appearance of Ω_k in S' with $S_x S_y$. If A is a member of the connected component X_i, then $A \notin Y$, so

procedure ScheduleLoops
input: a connected, consistent SDF graph G.
output: a valid unit blocking factor looped schedule $S_L(G)$ for G.

step 1: Use ϑ_2 to determine the nontrivial strongly connected components $Z_1, Z_2, \ldots, Z_s$ of G.
step 2: Cluster $Z_1, Z_2, \ldots, Z_s$ into the actors $\Omega_1, \Omega_2, \ldots, \Omega_s$ respectively, and denote the resulting graph by G'. This is an acyclic graph.
step 3: Apply ϑ_3 to G', and denote the resulting schedule by S'.
step 4:
  for $i = 1, 2, \ldots, s$
    Let G_z denote $subgraph(Z_i)$.
    Apply ϑ_1 to G_z.
    if $X, Y \subseteq Z_i$ are found such that X is subindependent of Y in G_z
      • Determine the connected components $X_1, X_2, \ldots, X_v$ and $Y_1, Y_2, \ldots, Y_w$ of $subgraph(X)$ and $subgraph(Y)$ respectively.
      • Recursively apply *ScheduleLoops* to construct
$S_x = (q_{G_z}(X_1)S_L(subgraph(X_1)))\ldots(q_{G_z}(X_v)S_L(subgraph(X_v)))$,
$S_y = (q_{G_z}(Y_1)S_L(subgraph(Y_1)))\ldots(q_{G_z}(Y_w)S_L(subgraph(Y_w)))$.
      • Replace the single appearance of Ω_i in S' with S_xS_y.
    else (G_z is tightly interdependent)
      • Apply ϑ_4 to obtain a valid schedule S_i for G_z.
      • Replace the single appearance of Ω_i in S' with S_i.
    end if
  end for
step 5: Output S' as $S_L(G)$.

Figure 5.2. Loose interdependence algorithms

$$appearances(A, S_xS_y)$$
$$= appearances(A, S_L(subgraph(X_i))).$$

Also, since A cannot be in any other strongly connected component besides Z_k, and since S' contains only one appearance of Ω_k, we have $appearances(A, S_L(G)) = appearances(A, S_xS_y)$. Thus, for $i = 1, 2, \ldots, v$,

$$(A \in X_i) \Rightarrow appearances(A, S_L(G)) = appearances(A, S_L(subgraph(X_i))).$$

By a similar argument, it can be shown that for $i = 1, 2, \ldots, w$, $(A \in Y_i) \Rightarrow appearances(A, S_L(G)) = appearances(A, S_L(subgraph(Y_i)))$.

We will apply a loose interdependence algorithm to derive nonrecursive necessary and sufficient conditions for the existence of a valid single appearance schedule. First, we introduce two useful lemmas.

Lemma 5.1: Suppose G is a connected, consistent SDF graph; A is an actor in G that is not contained in any tightly interdependent subgraph of G; and ζ is a loose interdependence algorithm. Then A appears only once in $S_\zeta(G)$, the schedule generated by ζ.

Proof: From Observation 5.1, if A is not contained in a nontrivial strongly connected component of G, the result is obvious, so we assume, without loss of generality, that A is in some nontrivial strongly connected component Z_1 of G. Since A is not contained in any tightly interdependent subgraph of G, $subgraph(Z_1)$ must be loosely interdependent, so ζ partitions Z_1 into X and Y, where X is subindependent of Y in $subgraph(Z_1)$. Let Z_1' denote that connected component of $subgraph(X)$ or $subgraph(Y)$ that contains A. From Observation 5.3,

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(Z_1'))).$$

From the assumptions of the lemma, all nontrivial strongly connected subgraphs of $subgraph(Z_1')$ that contain A are loosely interdependent. Thus, if A is contained in a nontrivial strongly connected component Z_2 of $subgraph(Z_1')$, then ζ will partition Z_2, and we will obtain a proper subset Z_2' of Z_1' such that

$$appearances(A, S_\zeta(subgraph(Z_1'))) = appearances(A, S_\zeta(subgraph(Z_2'))).$$

Continuing in this manner, we get a sequence $Z_1', Z_2', \ldots$ of subsets of $actors(G)$ such that each Z_i' is a proper subset of Z_{i-1}', A is contained in each Z_i', and

$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(Z_1'))) =$
$appearances(A, S_\zeta(subgraph(Z_2'))) = \ldots$

Since each Z_i' is a proper subset of its predecessor, we can continue this process only a finite number, say m, of times. Then $A \in Z_m'$, A is not contained in a nontrivial strongly connected component of $subgraph(Z_m')$, and

$$appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(Z_m'))).$$

But from Observation 5.1, $S_\zeta(subgraph(Z_m'))$ contains only one appearance of A. ■

Lemma 5.2: Suppose that G is a strongly connected, consistent SDF graph, $Y \subseteq actors(G)$ is subindependent in G, and Z is a strongly connected subset of $actors(G)$ such that $Y \cap Z \neq Z$ and $Y \cap Z \neq \varnothing$. Then $(Y \cap Z)$ is subindependent in $subgraph(Z)$.

Proof: Suppose that α is an edge directed from a member of $(Z - (Y \cap Z))$ to a member of $(Y \cap Z)$. By the subindependence of Y in G, $del(\alpha) \geq cns(\alpha)\mathbf{q}_G(snk(\alpha))$, and by Fact 4.5, $\mathbf{q}_G(snk(\alpha)) \geq \mathbf{q}_{subgraph(Z)}(snk(\alpha))$.

Thus, $del(\alpha) \geq cns(\alpha)\mathbf{q}_{subgraph(Z)}(snk(\alpha))$. Since this holds for any α directed from an actor in $(Z - (Y \cap Z))$ to an actor in $(Y \cap Z)$, we conclude that $(Y \cap Z)$ is subindependent in $subgraph(Z)$. ■

Corollary 5.1: Suppose that G is a strongly connected, consistent SDF graph, Z_1 and Z_2 are subsets of $actors(G)$ such that Z_1 is subindependent of Z_2 in G, and T is a tightly interdependent subgraph of G. Then $(actors(T) \subseteq Z_1)$ or $(actors(T) \subseteq Z_2)$.

Proof: (By contraposition). If $actors(T)$ has nonempty intersection with both Z_1 and Z_2, then from Lemma 5.2, $(actors(T) \cap Z_1)$ is subindependent in T, and thus, T is loosely interdependent. ■

Theorem 5.1: A nontrivial, strongly connected, consistent SDF graph G has a single appearance schedule if and only if every nontrivial strongly connected subgraph of G is loosely interdependent.

Proof: ($\Leftarrow$ direction). Suppose that every nontrivial strongly connected subgraph of G is loosely interdependent, and let ζ be any loose interdependence algorithm. Since no actor in G is contained in a tightly interdependent subgraph, it follows from Lemma 5.1 that $S_\zeta(G)$ is a single appearance schedule for G.

($\Rightarrow$ direction). Suppose that G has a single appearance schedule and that Z is a strongly connected subset of $actors(G)$ such that $|Z| > 1$. Set $Z_0 = G$. From Theorem 4.6, there exist $X_0, Y_0 \subseteq Z_0$ such that X_0 is subindependent of Y_0 in $subgraph(Z_0)$, and $subgraph(X_0)$ and $subgraph(Y_0)$ both have single appearance schedules. If X_0 and Y_0 do not both intersect Z, then Z is completely contained in some strongly connected component Z_1 of $subgraph(X_0)$ or $subgraph(Y_0)$. Theorem 4.6 can then be applied to partition Z_1 into X_1 and Y_1, and continue recursively in this manner until we obtain a strongly connected $Z_k \subseteq actors(G)$ with the following properties: there exist $X_k, Y_k \subseteq Z_k$ such that X_k is subindependent of Y_k in $subgraph(Z_k)$; $Z \subseteq Z_k$; and $(X_k \cap Z)$ and $(Y_k \cap Z)$ are both nonempty. From Lemma 5.2, $(X_k \cap Z)$ is subindependent in $subgraph(Z)$, so $subgraph(Z)$ must be loosely interdependent. ■

Corollary 5.2: Given a connected, consistent SDF graph G, any loose interdependence algorithm will obtain a single appearance schedule if one exists.

Proof: If a single appearance schedule for G exists, then from Theorem 5.1, G contains no tightly interdependent subgraphs. In other words, no actor in G is contained in a tightly interdependent subgraph of G. From Lemma 5.1, the schedule resulting from any loose interdependence algorithm contains only one appearance of each actor in G. ■

Thus, a loose interdependence algorithm always obtains an optimally compact solution when a single appearance schedule exists. When a single appearance schedule does not exist, strongly connected graphs are repeatedly decomposed until tightly interdependent subgraphs are found. In general, however, there may be more than one way to decompose $actors(G)$ into two parts so that one of the parts is subindependent of the other in G. Thus, it is natural to ask the following question: Given two distinct partitions $\{Z_1, Z_2\}$ and $\{Z_1', Z_2'\}$ of $actors(G)$ such that Z_1 is subindependent of Z_2 in G, and Z_1' is subindependent of Z_2' in G, is it possible that one of these partitions leads to a more compact schedule than the other? Fortunately, as will be shown in the remainder of this section, the answer to this question is "No". In other words, any two loose interdependence algorithms that use the same tight scheduling algorithm always lead to equally compact schedules. The key reason is that tight interdependence is an additive property.

Lemma 5.3: Suppose that G is a connected, consistent SDF graph, Y and Z are distinct strongly connected subsets of $actors(G)$ such that $(Y \cap Z) \neq \varnothing$, and $subgraph(Y)$ and $subgraph(Z)$ are both tightly interdependent. Then $subgraph(Y \cup Z)$ is tightly interdependent.

Proof: (By contraposition). Let $H = (Y \cup Z)$, and suppose that $subgraph(H)$ is loosely interdependent. Then there exist H_1 and H_2 such that H_1 is subindependent of H_2 in $subgraph(H)$. From $H_1 \cup H_2 = H = Y \cup Z$, and $Y \cap Z \neq \varnothing$, it is easily seen that H_1 and H_2 both have a nonempty intersection with Y, or they both have a nonempty intersection with Z. Without loss of generality, assume that $H_1 \cap Y \neq \varnothing$ and $H_2 \cap Y \neq \varnothing$. From Lemma 5.2, $(H_1 \cap Y)$ is subindependent in $subgraph(Y)$, and thus $subgraph(Y)$ is not tightly interdependent. ■

Lemma 5.3 implies that each SDF graph G has a unique set $\{T_1, T_2, \ldots, T_n\}$ of maximal tightly interdependent subgraphs such that $(i \neq j) \Rightarrow actors(T_i) \cap actors(T_j) = \varnothing$, and every tightly interdependent subgraph in G is contained in some T_i. We call each set $actors(T_i)$ a **tightly interdependent component** of G. It follows from Theorem 5.1 that G has a single appearance schedule if and only if G has no tightly interdependent components. Furthermore, since the tightly interdependent components are unique, the performance of a loose interdependence algorithm, with regards to schedule compactness, is not dependent on the particular subindependence partitioning algorithm, the component algorithm used to partition the loosely interdependent subgraphs. The following theorem develops this result.

Theorem 5.2: Suppose that G is a connected, consistent SDF graph, A is an actor in G, and ζ is a loose interdependence algorithm.

(a) If A is not contained in a tightly interdependent component of G, then A appears only once in $S_\zeta(G)$; and

(b) If A is contained in a tightly interdependent component X, then $appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(X)))$ — the number of appearances of A is determined entirely by the tight scheduling algorithm of ζ.

Proof: If A is not contained in a tightly interdependent component of G, then A is not contained in any tightly interdependent subgraph. Then from Lemma 5.1, $appearances(A, S_\zeta(G)) = 1$. Thus the proof of part (a) is complete.

Now suppose that A is contained in some tightly interdependent component X of G. If $X = actors(G)$, we are done. Otherwise, set $M_0 = actors(G)$, and thus $X \neq M_0$; by definition, tightly interdependent graphs are strongly connected, so X is contained in some strongly connected component Z of $subgraph(M_0)$.

If X is a proper subset of Z, then $subgraph(Z)$ must be loosely interdependent, since otherwise $subgraph(X)$ would not be a maximal tightly interdependent subgraph. Thus ζ partitions Z into V and W such that V is subindependent of W in $subgraph(Z)$. We set M_1 to be that connected component of $subgraph(V)$ or $subgraph(W)$ that contains A. Since V and W partition Z, M_1 is a proper subset of M_0. Also from Observation 5.3,

$$appearances(A, S_\zeta(subgraph(M_0))) = appearances(A, S_\zeta(subgraph(M_1))), \quad (5.1)$$

and from Corollary 5.1, $X \subseteq M_1$.

On the other hand, if $X = Z$, then we set $M_1 = X$. Since $X \neq M_0$, M_1 is a proper subset of M_0; from Observation 5.2, (5.1) holds, and trivially, $X \subseteq M_1$.

If $X \neq M_1$, then we can repeat the above procedure to obtain a proper subset M_2 of M_1 such that

$$appearances(A, S_\zeta(subgraph(M_1))) = appearances(A, S_\zeta(subgraph(M_2))),$$

and $X \subseteq M_2$. Continuing this process, we get a sequence $M_0, M_1, M_2, \ldots$. Since for each $i > 1$, M_i is a proper subset of its predecessor M_{i-1}, we cannot repeat this process indefinitely — eventually, for some $k \geq 1$, we will have $X = M_k$. But, by construction

$$\begin{aligned} appearances(A, S_\zeta(G)) &= appearances(A, S_\zeta(subgraph(M_0))) \\ &= appearances(A, S_\zeta(subgraph(M_1))) \\ &= \ldots = appearances(A, S_\zeta(subgraph(M_k))); \end{aligned}$$

and thus, $appearances(A, S_\zeta(G)) = appearances(A, S_\zeta(subgraph(X)))$. ■

Theorem 5.2 states that the tight scheduling algorithm is independent of the subindependence partitioning algorithm and vice-versa. Any subindependence partitioning algorithm guarantees that there is only one appearance for each actor outside the tightly interdependent components, and the tight scheduling algorithm completely determines the number of appearances for actors inside the tightly interdependent components. For example, if we develop a new subindependence partitioning algorithm that is more efficient in some way (for example, it is faster or minimizes the memory required to implement buffering), we can replace it for any existing subindependence partitioning algorithm without changing the com-

pactness of the resulting schedules — we don't need to analyze its interaction with the rest of the loose interdependence algorithm. Similarly, if we develop a new tight scheduling algorithm that schedules any tightly interdependent graph more compactly than the existing tight scheduling algorithm, we are guaranteed that using the new algorithm instead of the old one will lead to more compact schedules overall.

The complexity of a loose interdependence algorithm ζ depends on its subindependence partitioning algorithm ζ_{sp}, strongly connected components algorithm ζ_{sc}, acyclic scheduling algorithm ζ_{as}, and tight scheduling algorithm ζ_{ts}. From Definition 5.2, we see that ζ_{ts} is applied exactly once for each tightly interdependent component. For example, the algorithm specified in Section 3.2, *ConstructValidSchedule*, can be used as the tight scheduling algorithm. If this algorithm is applied to a tightly interdependent component X, it runs in time that is linear in the total number of invocations in a minimal schedule period of $subgraph(X)$. That is, the running time is $O(I_X f_i' f_o' + |E_X|)$, where E_X is the set of edges in $subgraph(X)$,

$$I_X = \sum_{A \in X} \mathbf{q}_{subgraph(X)}(A),$$

and f_i' (f_o') is the maximum over all actors in $subgraph(X)$ of the number of input (output) edges that are incident to any actor.

Thus, if ζ_{ts} is algorithm *ConstructValidSchedule* (from Section 3.2) and ζ is applied to an SDF graph G, the total time that ζ_{ts} accounts for is $O(I_G f_i f_o + |E_G|)$, where

$$I_G = \sum_{A \in actors(G)} \mathbf{q}_G(A),$$

and f_i (f_o) is the maximum over all actors in G of the number of input (output) edges that are incident to any actor.

The other component algorithms, ζ_{sc}, ζ_{as}, and ζ_{sp}, are successively applied to decompose an SDF graph, and the process is repeated until all tightly interdependent components are found. In the worst case, each decomposition step isolates a single actor from the current n-actor subgraph, and the decomposition must be recursively applied to the remaining $(n-1)$-actor subgraph. Thus, if G denotes the input SDF graph, then ζ performs $|actors(G)|$ decomposition steps

in the worst case. Tarjan's algorithm [Tarj72] allows the strongly connected components of G to be found in $O(m)$ time, where

$$m = max(\{|actors(G)|, |edges(G)|\}).$$

Hence ζ_{sc} can be chosen to be linear, and since at most $|actors(G)| \le m$ decomposition steps are required, the total time that such a ζ_{sc} accounts for in ζ is $O(m^2)$. Finally Section 4.6 described a simple linear-time algorithm that constructs a single appearance schedule for an acyclic graph. Thus ζ_{as} can also be chosen such that its total time is also $O(m^2)$.

We have specified ζ_{sp}, ζ_{sc}, ζ_{as}, and ζ_{ts} such that the resulting loose interdependence algorithm ζ has worst-case running time that is $O(m^2 + I_G f_i f_o)$, where $m = max(\{|actors(G)|, |edges(G)|\})$.

Note that this worst-case estimate is conservative — in practice, usually only a few decomposition steps are required to fully schedule a strongly connected subgraph, while the estimate assumes $|actors(G)|$ steps. Furthermore, a more accurate expression for the total time that the tight scheduling algorithm accounts for is $O(I_T f_i f_o + |E_T|)$ where

$$I_T = \sum_{i=1}^{p} \sum_{A \in actors(T_i)} \mathbf{q}_{T_i}(A),$$

$$E_T = \bigcup_{i=1}^{p} edges(T_i),$$

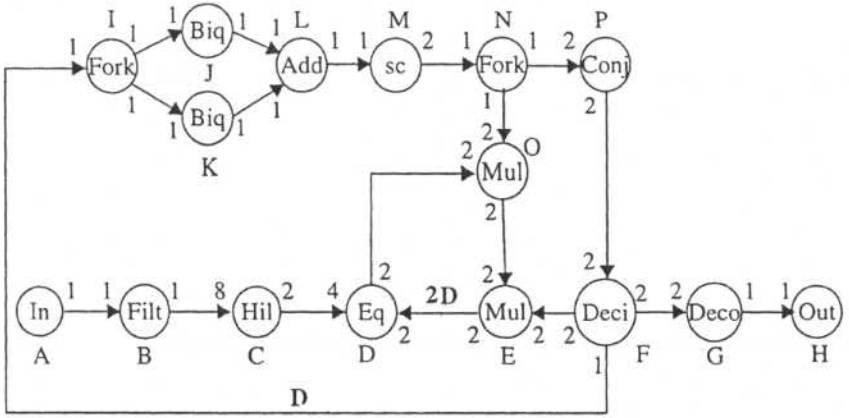
and $T_1, T_2, \ldots, T_p$ are the subgraphs associated with the tightly interdependent components of G. When the tightly interdependent components form only a small part of G this bound will be much tighter than the $O(I_G f_i f_o + m^2)$ bound.

5.2 Modem Example

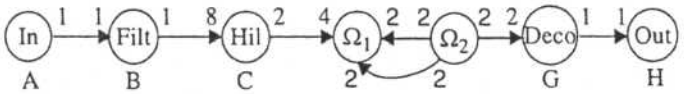
Figure 5.3(a) shows an SDF implementation of a modem taken from [Lee86]. The repetitions vector is given by

$$\mathbf{q}(A, B, \ldots, P) = [16, 16, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1]^T.$$

There are 2 strongly connected components, and they are the cycle corresponding to actors D, O, E and the cycles corresponding to actors F, I, J, K, L, M, N, P. These strongly connected components are clustered to give an acyclic graph as depicted in Figure 5.3(b). A possible single appearance schedule for this graph is



a)



b)

Figure 5.3. a) A block diagram of 2400 bps, 600 baud modem. b) Acyclic graph after clustering the strongly connected components in a).

$(16A)(16B)(2C)\Omega_2\Omega_1 GH$. Both of the strongly connected components have subindependent partitions. For the first strongly connected component, the partition is given by $\{D\}$ and $\{O, E\}$. Since both of these subgraphs are acyclic, the recursive application of a loose interdependence algorithm (procedure *Schedule-Loops*) terminates by applying ϑ_3 to each of the partitions, yielding the single appearance schedule DOE for this strongly connected component. Similarly, the second strongly connected component also has a subindependent partition and the recursive application of the loose interdependence framework terminates by producing the schedule $IJKLM(2N)PF$ (of-course, it may terminate with other single appearance schedules as well since there are many other ones for this graph). These schedules are substituted into the top-level schedule to give the single appearance schedule for the entire graph:

$$(16A)(16B)(2C)IJKLM(2N)PFDOEGH.$$

5.3 Clustering in a Loose Interdependence Algorithm

As was discussed in Section 4.3, clustering subgraphs — grouping subgraphs so that they are invoked as single units — can be used to guide a scheduler toward more efficient schedules. However, certain clustering decisions conflict with code-space minimization goals, and thus if any clustering is to be incorporated into a loose interdependence algorithm, then the possible degradation on code-compaction potential should be considered. In this section, a useful clustering technique is developed for increasing the frequency of data transfers that occur through machine registers rather than memory, and this technique is proven not to interfere with the code compactness potential of a loose interdependence algorithm — that is, this clustering preserves the properties of loose interdependence algorithms discussed in the previous section.

Figure 5.4 illustrates two ways in which arbitrary clustering decisions can conflict with code compactness objectives. Observe that the SDF graph in Figure 5.4(a) is acyclic, so it must have a single appearance schedule. Figure 5.4(b)

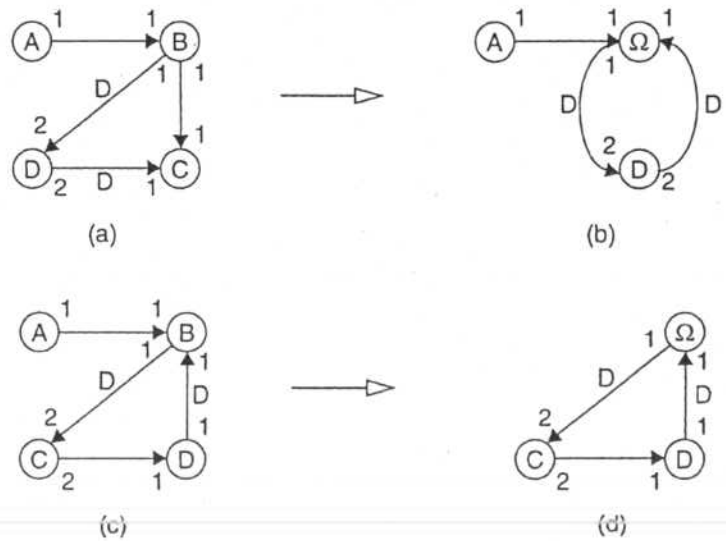


Figure 5.4. Examples of clustering decisions that conflict with code compactness goals.

shows the hierarchical SDF graph that results from clustering actors B and C in Figure 5.4(a) into the single actor Ω. It is easily verified that in Figure 5.4(b), $subgraph(\{\Omega, D\})$ is tightly interdependent. Thus, the clustering of B and C in Figure 5.4(a) cancels the existence of a single appearance schedule.

In Figure 5.4(c), $subgraph(\{B, C, D\})$ is a tightly interdependent component and actor A is not contained in any tightly interdependent subgraph. From Theorem 5.2, we know that any loose interdependence algorithm will schedule the graph of Figure 5.4(c) in such a way that A appears only once. Now observe that the hierarchical SDF graph that results from clustering A and B, shown in Figure 5.4(d), is a tightly interdependent graph. It can be verified that the most compact minimal periodic schedule for this graph is $\Omega C(2D)\Omega$, which leads to the schedule $ABC(2D)AB$ for Figure 5.4(c). By increasing the extent of the tightly interdependent component $subgraph(\{B, C, D\})$ to subsume actor A, this clustering decision increases the minimum number of appearances of A in the final schedule.

Thus, we see that a clustering decision can conflict with optimal code compactness if it introduces a new tightly interdependent component or extends an existing tightly interdependent component. In this section, a clustering technique of practical use is presented and it is proven that it neither extends nor introduces tight interdependence. The clustering technique and its compatibility with loose interdependence algorithms is summarized by Fact 5.2 below. This fact is an immediate corollary of Theorem 5.3, which will be presented later in this section. Establishing Theorem 5.3 is the main topic of the remainder of this section.

Fact 5.2: Clustering two adjacent actors A and B in an SDF graph does not introduce or extend a tightly interdependent component if (a) Neither A nor B is contained in a tightly interdependent component; (b) At least one edge directed from A to B has zero delay; (c) A and B are invoked the same number of times in a periodic schedule; *and* (d) B has no predecessors other than A or B. Since these conditions are sufficient but not necessary, they provide a conservative guide for clustering.

We motivate the clustering technique with the example shown in Figure 5.5. The repetitions vector for the SDF graph in Figure 5.5(a) is $\mathbf{q}(A, B, C, D, E) = (1, 10, 10, 10, 1)^T$, and one valid single appearance schedule for this graph is $(10C)(10D)EA(10B)$. This schedule is inefficient with regards to buffering. Due to the schedule loop that specifies ten successive invocations of

actor C, the data transfers between C and D cannot take place in machine registers, and 10 units of memory are required to implement the edge $C \rightarrow D$. However, observe that the four conditions of Fact 5.2 all hold for the adjacent pairs $\{C, D\}$ and $\{A, E\}$. Thus, we can cluster these pairs without cancelling the existence of a single appearance schedule. The hierarchical SDF graph that results from this clustering is shown in Figure 5.5(b); this graph leads to the valid single appearance schedule $(10\Omega_2)\Omega_1(10B) \Rightarrow (10CD)EA(10B)$. In this second schedule, each token produced by C is consumed by D in the same loop iteration, so all of the transfers between C and D can occur through a single machine register. Thus, the clustering of C and D saves 10 units of memory for the data transfers between C and D, and it allows these transfers to be performed through a register rather than memory, which will usually result in faster code.

When it is not ambiguous, we will use the following additional notation in the development of this section.

Definition 5.3: Let G be an SDF graph, and suppose that we cluster a subset W of actors in G. We will refer to the resulting hierarchical SDF graph as G', and we will refer to the actor in G' into which W has been clustered as Ω. For each edge α in G that is not contained in $subgraph(W, G)$, we denote the corresponding edge in G' by α'. Finally, if $X \subseteq actors(G)$, we denote the corresponding subset of $actors(G')$ as X'. That is, X' contains all members of X that are not in W, and if X contains one or more members of W, then X' also contains Ω.

For example if G is the SDF graph in Figure 5.4(a), $W = \{B, C\}$, and α and β respectively denote $B \rightarrow D$ and $A \rightarrow B$, then we denote the graph in Fig-

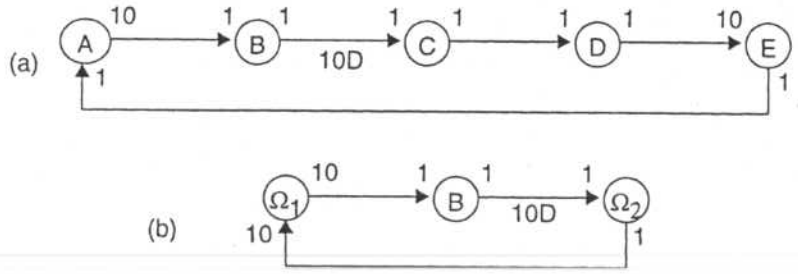


Figure 5.5. An example of clustering to increase the frequency of data transfers that occur through registers rather than memory.

ure 5.4(b) by G', and in G', we denote $\Omega \to D$ by α' and $A \to \Omega$ by β'. Also, if $X = \{A, B\}$ then $X' = \{A, \Omega\}$.

Lemma 5.4: Suppose that G is a strongly connected, consistent SDF graph, and X_1 and X_2 form a partition of $actors(G)$ such that X_1 is subindependent of X_2 in G. Also, suppose that A and B are actors in G such that $A, B \in X_1$ or $A, B \in X_2$. If we cluster $W = \{A, B\}$, then the resulting SDF graph G' is loosely interdependent.

Proof: Let Φ denote the set of edges in G that are directed from an actor in X_2 to an actor in X_1, and let Φ' denote the set of edges in G' that are directed from an actor in X_2' to an actor in X_1'. Since $subgraph(\{A, B\})$ does not contain any edges in Φ, it follows that $\Phi' = \{\alpha' | \alpha \in \Phi\}$. From Fact 4.7, we have that for all α', $\mathbf{q}_{G'}(snk(\alpha'))cns(\alpha') = \mathbf{q}_G(snk(\alpha))cns(\alpha)$. Now since X_1 is subindependent of X_2 in G, for all $\alpha \in \Phi$, $del(\alpha) \geq \mathbf{q}_G(snk(\alpha))cns(\alpha)$. It follows that for all $\alpha' \in \Phi'$, $del(\alpha') \geq \mathbf{q}_{G'}(snk(\alpha'))cns(\alpha')$, and we conclude that X_1' is subindependent of X_2' in G'. But, by construction, X_1' and X_2' partition $actors(G')$; thus, G' is loosely interdependent. ■

Lemma 5.5: Suppose that G is a connected, consistent SDF graph, Z is a proper subset of $actors(G)$, $A_1 \in Z$, and A_2 is an actor that is contained in $actors(G)$ but not in Z such that

(1) A_2 is not adjacent to any member of $(Z - \{A_1\})$, and

(2) for some positive integer k, $\mathbf{q}_G(A_2) = k\mathbf{q}_G(A_1)$.

If we cluster $W = \{A_1, A_2\}$ in G, then $subgraph(Z - \{A_1\} + \{\Omega\}, G')$ is isomorphic to $subgraph(Z, G)$.

As a simple illustration, consider again the clustering example of Figure 5.4(c) and Figure 5.4(d). Let G and G' respectively denote the graphs of Figure 5.4(c) and Figure 5.4(d), and let $Z = \{B, C\}$, $A_1 = B$, and $A_2 = A$. Then $(Z - \{A_1\} + \{\Omega\}) = \{C, \Omega\}$, and clearly, $subgraph(\{C, \Omega\}, G')$ is isomorphic to $subgraph(\{B, C\}, G)$.

Proof of Lemma 5.5: Let $X = subgraph(Z - \{A_1\} + \{\Omega\}, G')$, let Φ denote the set of edges in $subgraph(Z, G)$, and let Φ' denote the set of edges in X. From (1), every edge in X has a corresponding edge in $subgraph(Z, G)$, and vice-versa, and thus $\Phi' = \{\alpha' | \alpha \in \Phi\}$. Now, from the definition of clustering a subgraph, we know that $prd(\alpha') = prd(\alpha)$ for any edge $\alpha \in \Phi$ such that $src(\alpha) \neq A_1$. If $src(\alpha) = A_1$ then α is replaced by α' with $src(\alpha') = \Omega$, and

$$prd(\alpha') = prd(\alpha)\mathbf{q}_G(A_1)/gcd(\{\mathbf{q}_G(A_1), \mathbf{q}_G(A_2)\}).$$

But,

$$gcd(\{\mathbf{q}_G(A_1), \mathbf{q}_G(A_2)\}) = gcd(\{\mathbf{q}_G(A_1), k\mathbf{q}_G(A_1)\}) = \mathbf{q}_G(A_1),$$

so $prd(\alpha') = prd(\alpha)$. Thus $prd(\alpha') = prd(\alpha)$ for all $\alpha \in \Phi$.

Similarly, we can show that $cns(\alpha') = cns(\alpha)$ for all $\alpha \in \Phi$. Thus, the mappings $f_1 : Z \to actors(X)$ and $f_2 : \Phi \to \Phi'$ defined by $f_1(A) = A$ if $A \neq A_1$, $f_1(A_1) = \Omega$; and $\forall \alpha, f_2(\alpha) = \alpha'$ demonstrate that $subgraph(Z, G)$ is isomorphic to X. ■

Lemma 5.6: Suppose that G is a consistent, strongly connected SDF graph and Z is a strongly connected subset of actors in G such that $q_G(Z) = 1$. Suppose Z_1 and Z_2 form a partition of Z such that Z_1 is subindependent of Z_2 in $subgraph(Z, G)$. Then Z_1 is subindependent of Z_2 in G.

Proof: For each edge α directed from a member of Z_2 to a member of Z_1, we have

$$del(\alpha) \geq \mathbf{q}_{subgraph(Z)}(snk(\alpha))cns(\alpha).$$

From Fact 4.5, $\mathbf{q}_{subgraph(Z)}(A) = \mathbf{q}_G(A)$ for all $A \in Z$. Thus, for all edges α in $subgraph(Z)$, $\mathbf{q}_{subgraph(Z)}(snk(\alpha))cns(\alpha) = \mathbf{q}_G(snk(\alpha))cns(\alpha)$, and we conclude that Z_1 is subindependent of Z_2 in G. ■

Lemma 5.7: Suppose that G is a consistent, strongly connected SDF graph, A and B are distinct actors in G, and $W = \{A, B\}$ forms a proper subset of $actors(G)$. Suppose also that the following four conditions all hold:

(1) Neither A nor B is contained in a tightly interdependent subgraph of G.

(2) There is at least one edge directed from A to B that has zero delay.

(3) B has no predecessors other than A or B.

(4) $\mathbf{q}_G(B) = k\mathbf{q}_G(C)$ for $k \in \{1, 2, 3, \ldots\}$, and for some $C \in actors(G)$ such that $C \neq B$.

Then the SDF graph G' that results from clustering W in G is loosely interdependent.

Proof: From (1), G must be loosely interdependent, so there exist subsets Z_1 and Z_2 of $actors(G)$ such that Z_1 and Z_2 partition $actors(G)$, and Z_1 is subindependent of Z_2 in G. If $A, B \in Z_1$ or $A, B \in Z_2$, then from Lemma 5.4, we are done. Now, condition (2) precludes the scenario $((B \in Z_1)$ **and** $(A \in Z_2))$, so the

only remaining possibility is $((A \in Z_1)$ **and** $(B \in Z_2))$. There are two sub-cases to consider here:

(i). B is not the only member of Z_2. Then from (3), $(Z_1 + \{B\})$ is subindependent of $(Z_2 - \{B\})$. But $A, B \in Z_1 + \{B\}$, so Lemma 5.4 again guarantees that G' is loosely interdependent.

(ii). $Z_2 = \{B\}$. Thus, we have Z_1 is subindependent of $\{B\}$, so

$$\forall(\alpha \in \{\hat{\alpha} \in edges(G) | snk(\hat{\alpha}) \neq B\}),$$
$$(src(\alpha) = B) \Rightarrow del(\alpha) \geq TNSE(\alpha, G). \tag{5.2}$$

Also, since $C \in Z_1$, we have from (4) that

$$\begin{aligned} q_G(Z_1) &= gcd(\{\mathbf{q}_G(N) | (N \in Z_1)\}) \\ &= gcd(\{\mathbf{q}_G(N) | (N \in Z_1)\} \cup \{k\mathbf{q}_G(C)\}) \\ &= gcd(\{\mathbf{q}_G(N) | (N \in Z_1)\} \cup \{\mathbf{q}_G(B)\}) \\ &= gcd(\{\mathbf{q}_G(N) | N \in actors(G)\}) = 1. \end{aligned}$$

That is,

$$q_G(Z_1) = 1. \tag{5.3}$$

Now if Z_1 is not strongly connected, then it has a proper subset Y such that there are no edges directed from a member of $(Z_1 - Y)$ to a member of Y. Furthermore, from condition (3), $A \notin Y$. This is true because if Y contained A, then there would be no path directed from a member of $(Z_1 - Y)$ to B, and thus G would not be strongly connected. Thus, $A \in (Z_1 - Y)$, and there are no edges directed from a member of $(Z_1 - Y)$ to a member of Y. So all edges directed from a member of $(Z_1 - Y + \{B\})$ to Y have actor B as their source. From (5.2), it follows that Y is subindependent of $(Z_1 - Y + \{B\})$ in G. Now, $A, B \in (Z_1 - Y + \{B\})$, so applying Lemma 5.4, we conclude that G' is loosely interdependent.

If Z_1 is strongly connected, we know from condition (1) that there exists a partition X_1, X_2 of Z_1 such that X_1 is subindependent of X_2 in $subgraph(Z_1)$. From (5.3) and Lemma 5.6, X_1 is subindependent of X_2 in G. Now if $A \in X_1$, then from condition (3), $\{B\}$ is subindependent of X_2 in G, so from Fact 4.10(a), $(X_1 \cup \{B\})$ and X_2 constitute a subindependent partition of G. Applying Lemma 5.4, we see that G' is loosely interdependent. On the other hand, suppose that $A \in X_2$. Then from (5.2), we know that X_1 is subindependent of $\{B\}$ in G. From Fact 4.10(b), it follows that X_1 and $(X_2 \cup \{B\})$ constitute a subindependent partition of G, so again we can apply Lemma 5.4 to conclude that G' is loosely interdependent. ■

Theorem 5.3: Suppose that G is a consistent, connected SDF graph, A and B are distinct actors in G such that B is a successor of A, and $W = \{A, B\}$ is a proper subset of $actors(G)$. Suppose also that the following four conditions all hold:

(1) Neither A nor B is contained in a tightly interdependent component of G.

(2) At least one edge directed from A to B has zero delay.

(3) For some positive integer k, $\mathbf{q}_G(B) = k\mathbf{q}_G(A)$.

(4) Actor B has no predecessors other than A or B.

Then the tightly interdependent components of G' are the same as the tightly interdependent components of G.

Proof: Observe that all subgraphs in G that do not contain A nor B are not affected by the clustering of W, and thus it suffices to show that all strongly connected subgraphs in G' that contain Ω are loosely interdependent. So we suppose that Z' is a strongly connected subset of actors in G' that contains Ω, and we let Z denote the corresponding subset of actors in G; that is $Z = Z' - \{\Omega\} + \{A, B\}$. Now, in $subgraph(Z', G')$, suppose that there is a cycle consisting of Ω and two other actors, C and D. From condition (4), this implies that there is a cycle in G containing A, C, D, and possibly B. The two possible ways in which a cycle in G introduces a cycle consisting of Ω in G' are illustrated in Figure 5.6(a) and (b); the situation in Figure 5.6(c) cannot arise because of condition (4).

Now in $subgraph(Z', G')$, if one or more of the cycles that pass through Ω correspond to Figure 5.6(a), then Z must be a strongly connected subset in G. Otherwise, all of the cycles involving Ω correspond to Figure 5.6(b), so $(Z - \{B\})$ is strongly connected, and from condition (4), no member of $(Z - \{A, B\})$ is adjacent to B. In the former case, Lemma 5.7 immediately yields the loose interdependence of $subgraph(Z', G')$.

In the latter case, Lemma 5.5 guarantees that $subgraph(Z - \{B\}, G)$ is isomorphic to $subgraph(Z', G')$. Since $A \in (Z - \{B\})$, and since from condition (1), A is not contained in any tightly interdependent subgraph of G, it follows that $subgraph(Z', G')$ is loosely interdependent. ■

If we assume that the input SDF graph has a single appearance schedule, then we can ignore condition (1). From our observations, this is a valid assumption for a large class of practical SDF graphs. Also, condition (3) can be verified by examining any single edge directed from A to B; if α is an edge directed from A to B, then condition (3) is equivalent to $prd(\alpha) = kcns(\alpha)$.

The following corollary assures us that when applying Theorem 5.3, no further checks are necessary to determine whether the clustering of A and B introduces deadlock.

Corollary 5.3: Assume the hypotheses of Theorem 5.3, including conditions (1) through (4). Then G' is not deadlocked.

Proof: (By contraposition). If G' is deadlocked, then there exists a fundamental cycle in G' whose associated graph G_f is deadlocked. By the definition of tight interdependence, G_f is tightly interdependent, so $actors(G_f)$ is contained in some tightly interdependent component X of G'. Thus, Theorem 5.3 guarantees that $subgraph(X, G')$ is a tightly interdependent subgraph of G, and hence that the deadlocked graph G_f is contained in G. It follows that G is deadlocked, and G is not a consistent SDF graph. ■

Under the assumption that the input SDF graph has a single appearance schedule, the clustering process defined by Theorem 5.3 requires only *local* dataflow information, and thus it can be implemented very efficiently. If the assumption that a single appearance schedule exists is wrong, then we can always undo the clustering decisions. Finally, in addition to making buffering more efficient,

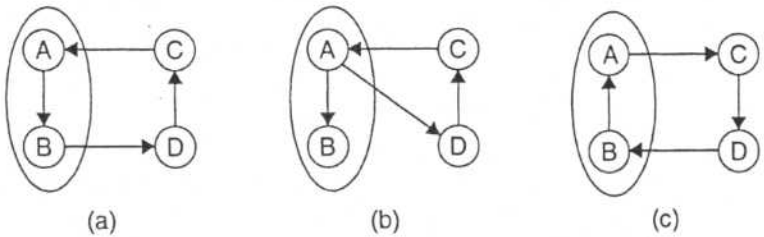


Figure 5.6. An illustration of how a cycle containing Ω originates in G' for Theorem 5.3. The two possible scenarios are shown in (a) and (b); (c) will not occur due to condition (4). SDF parameters on the edges have not been assigned because they are not relevant to the introduction of cycles.

the clustering process provides a fast way to reduce the size of an SDF graph without cancelling the existence of a single appearance schedule. When used as a preprocessing technique, this can sharply reduce the execution time of a loose interdependence algorithm.

5.4 Relation to Vectorization

The techniques developed in Section 5.1 in this book are related to techniques for transforming serial procedural programs into programs that are suitable for vector processors. Vector processors are computers that have special operations, called *vector instructions*, for operating on arrays of data. For example, in a vector processor, the following loop can be implemented by a single vector instruction:

```
DO 10 I = 1, 100
   X(I) = Y(I+10) + Z(I+20)
10 CONTINUE
```

A common syntax for the vector instruction corresponding to this loop is

```
X(1:100) = Y(11:110) + Z(21:120)
```

In a vector instruction, the computations of the components of the result vector are independent of one another, so deep pipelines can be employed without any hazards [Kogg81]. Also with a vector instruction, the number of instructions that must be fetched and decoded is reduced; interleaved memories can be exploited to reduce the average time required to read an operand from memory; and the pipeline hazards arising from the loop branch in the original (unvectorized) loop are eliminated [Henn90]. Often, as a consequence of upgrades in computing resources, programs written for conventional scalar processors must be ported to vector processors. Also, from the programmer's viewpoint, it is often more natural or convenient to write serial programs without worrying about efficiently utilizing vector instructions. These considerations have motivated the study of automatic techniques for vectorizing serial procedural programs.

Vectorization algorithms normally operate on a data structure called a *dependence graph*. The dependence graph of a procedural program segment is a directed graph in which each vertex corresponds to a statement of the program. If v_1 and v_2 are vertices of a dependence graph and s_1 and s_2 are, respectively, the corresponding statements, then there is an edge directed from v_1 to v_2 if it has been determined that some invocation of s_2 is dependent on an invocation of s_1;

that is, there exist invocations i_1 and i_2 of s_1 and s_2, respectively, such that executing i_2 before i_1 may be inconsistent with the semantics of the original program.

Unlike the precedence relationships specified by an SDF graph, the dependences in a dependence graph cannot always be determined exactly at compile-time. This is because the programming languages to which dependence graphs are applied are based on more general models of computation than SDF. For example, consider the following FORTRAN code segment in which the value of the variable X is not known at compile-time.

```
        DO 10 I = 1, X
s1 :        A(I) = 1
s2 :        B(I) = A(100 - I)
        10 CONTINUE
```

Here, s_2 depends on s_1 if and only if $X \geq 50$. Unless it is known that the value of X will definitely be less than 50, there is a dependence graph edge directed from the vertex corresponding to s_1 to the vertex corresponding to s_2.

Another significant difference between SDF graphs and dependence graphs is that SDF graph edges specify iteration implicitly — through mismatches in the production and consumption parameters — whereas with dependence graphs, the repetition of statements results from control-flow structure that is specified explicitly in the corresponding program. With SDF graphs, no control-flow structure exists a-priori, and we must construct one carefully with regards to the available memory in the target processor before proceeding with other scheduling optimizations. Once the control-flow has been specified for an SDF graph, and code blocks for each actor have been inlined, dependence graphs can be constructed and dependence graph analysis can be applied to further optimize the target program. However, the construction of the initial control-flow structure is a crucial step, and we expect that failure in this step is generally difficult to overcome through post-optimization. For example, a study by How [How90] confirmed that pattern matching on a schedule designed for minimum buffer memory requirement does not acceptably minimize the code size. When compiling an SDF graph, the scheduling framework of Section 5.1 can be applied first. If the resulting target program fits within the available processor memory, then post-optimization techniques, such as those that apply dependence graphs, loop unrolling

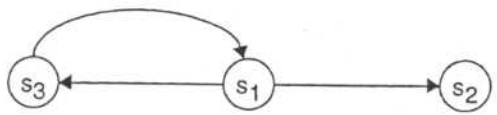
[Dong79], or reorganizing the loop structure to improve memory access locality [Wolf91], can be applied until the remaining memory is exhausted.

The vectorization problem is similar in structure to the problem of constructing compact looped schedules for SDF graphs since just as strongly connected components in an SDF graph can limit looping opportunities, cycles in a dependence graph limit vectorization. Vectorization is most commonly applied to the innermost loop of a group of nested loops. If the dependence graph for the inner loop is acyclic, then each statement can be vectorized provided that a matching vector instruction exists. If cycles are present, then they are carefully analyzed to see if they can be ignored or if transformations can be applied to eliminate them [Wolf89].

A common tool for vectorization is the *loop distribution* transformation, which was introduced by Muraoka in [Mura71]. In loop distribution, the body of a loop is partitioned into segments, and a separate loop is created for each segment. As an example of loop distribution, and how it can be applied to vectorization, consider the FORTRAN loop below.

```
        DO 10 I = 1, 10
s1 :        A(I) = B(I) + C(I - 1)
s2 :        D(I) = 2 * A(I)
s3 :        C(I) = A(I) + 5
        10 CONTINUE
```

The dependence graph for this loop is:



We see that s_1 and s_3 form a dependence graph cycle, and that s_2 is not part of any cycle. We can replace the loop with one loop that spans the s_1-s_3 cycle and a second loop for s_2, which can be vectorized. The transformed program that results from this combination of loop distribution and vectorization is shown below.

```
DO 10 I = 1, 10
   A(I) = B(I) + C(I - 1)
   C(I) = A(I) + 5
10 CONTINUE
D(1:10) = 2 * A(1:10)
```

We see that this method of transformation bears similarities with the loose interdependence scheduling framework.

If the target processor has multidimensional vector instructions available, then it may be desirable to vectorize across multiple nested loops. Nested loop vectorization is the form of vectorization that is most closely related to the techniques developed in Section 5.1 of this book. Two main approaches to nested loop vectorization have emerged — the *outside-in* vectorization of Allen and Kennedy [Alle87], and the *inside-out* vectorization of Muraoka [Mura71]. Respectively, the relationship between these two techniques is somewhat analogous to the differences between the loose interdependence scheduling framework and the minimum activation scheduling method of COSSAP described in Section 1.8.3.

Suppose that $L_1, L_2, \ldots, L_n$ is a sequence of perfectly nested FORTRAN loops; that is, there are no statements between the loops. Suppose that L_1 is the outermost loop, L_2 is the next outermost loop, and so on. In outside-in vectorization, the L_i's are traversed starting with the outermost loop and working inward. First, the dependence graph for $L_1, L_2, \ldots, L_n$ is examined, and loop distribution is applied to isolate strongly connected components and vectorizeable statements. Then, for each strongly connected component, the L_1 loop is fixed and the dependence graph for $L_2, L_3, \ldots, L_n$ is examined. Again, loop distribution is applied, and the method continues recursively on each strongly connected component of the dependence graph for the $L_2, L_3, \ldots, L_n$ combination.

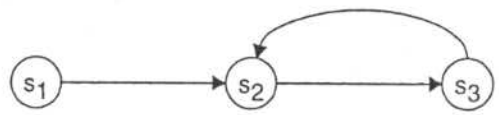
For example, consider the nested loops below.

```
      DO 10 I = 1, 100
         DO 20 J = 1, 100
s1 :        A(I, J) = X(I, J) + Y(I, J)
s2 :        B(I, J) = A(I, J) + C(I - 1, J)
s3 :        C(I, J) = B(I, J) * 6
```



```
   20 CONTINUE
   10 CONTINUE
```

The associated dependence graph is:



Since s_1 is not part of a dependence cycle, it is isolated and vectorized, and this results in the transformed program below.

```
      A(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
      DO 10 I= 1, 100
         DO 20 J = 1, 100
s2 :        B(I, J) = A(I, J) + C(I - 1, J)
s3 :        C(I, J) = B(I, J) * 6
   20 CONTINUE
   10 CONTINUE
```

Next, the dependence graph for the inner loop is examined:

Since no dependence graph cycles exist, the inner loop can be vectorized, and the final result of applying outside-in vectorization is:

```
      A(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
      DO 10 I= 1, 100
         B(I,1:100) = A(I,1:100) + C(I-1,1:100)
         C(I,1:100) = B(I, 1:100) * 6
   10 CONTINUE
```

```
DO 10 I = 1, 10
   A(I) = B(I) + C(I - 1)
   C(I) = A(I) + 5
10 CONTINUE
D(1:10) = 2 * A(1:10)
```

We see that this method of transformation bears similarities with the loose interdependence scheduling framework.

If the target processor has multidimensional vector instructions available, then it may be desirable to vectorize across multiple nested loops. Nested loop vectorization is the form of vectorization that is most closely related to the techniques developed in Section 5.1 of this book. Two main approaches to nested loop vectorization have emerged — the *outside-in* vectorization of Allen and Kennedy [Alle87], and the *inside-out* vectorization of Muraoka [Mura71]. Respectively, the relationship between these two techniques is somewhat analogous to the differences between the loose interdependence scheduling framework and the minimum activation scheduling method of COSSAP described in Section 1.8.3.

Suppose that $L_1, L_2, \ldots, L_n$ is a sequence of perfectly nested FORTRAN loops; that is, there are no statements between the loops. Suppose that L_1 is the outermost loop, L_2 is the next outermost loop, and so on. In outside-in vectorization, the L_i's are traversed starting with the outermost loop and working inward. First, the dependence graph for $L_1, L_2, \ldots, L_n$ is examined, and loop distribution is applied to isolate strongly connected components and vectorizeable statements. Then, for each strongly connected component, the L_1 loop is fixed and the dependence graph for $L_2, L_3, \ldots, L_n$ is examined. Again, loop distribution is applied, and the method continues recursively on each strongly connected component of the dependence graph for the $L_2, L_3, \ldots, L_n$ combination.

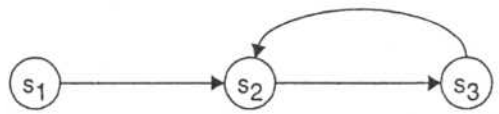
For example, consider the nested loops below.

```
      DO 10 I = 1, 100
         DO 20 J = 1, 100
s1 :        A(I, J) = X(I, J) + Y(I, J)
s2 :        B(I, J) = A(I, J) + C(I - 1, J)
s3 :        C(I, J) = B(I, J) * 6
```



```
   20 CONTINUE
   10 CONTINUE
```

The associated dependence graph is:



Since s_1 is not part of a dependence cycle, it is isolated and vectorized, and this results in the transformed program below.

```
      A(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
      DO 10 I= 1, 100
         DO 20 J = 1, 100
s2 :        B(I, J) = A(I, J) + C(I - 1, J)
s3 :        C(I, J) = B(I, J) * 6
      20 CONTINUE
      10 CONTINUE
```

Next, the dependence graph for the inner loop is examined:

Since no dependence graph cycles exist, the inner loop can be vectorized, and the final result of applying outside-in vectorization is:

```
A(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
DO 10 I= 1, 100
   B(I,1:100) = A(I,1:100) + C(I-1,1:100)
   C(I,1:100) = B(I, 1:100) * 6
10 CONTINUE
```

This approach bears resemblance to the scheduling framework of loose interdependence algorithms. When scheduling SDF graphs, the outermost loop corresponds to a singe period of the periodic schedule. The strongly connected components of the SDF graph are isolated by the clustering process of step 2 in Figure 5.2. Then, for each strongly connected component, we focus on the next inner loop nesting level of the target program by examining the interdependencies within a minimal schedule period for the given strongly connected component and attempting to find a subindependent partition. Just as some dependence graph edges disappear as we descend the nesting levels of a group of nested loops, SDF graph edges can become "ignorable" as a loose interdependence algorithm recursively decomposes strongly connected components of an SDF graph. Given a consistent, connected SDF graph G, an edge α does not impose precedence constraints within a minimal schedule period for G if and only if $del(\alpha) \geq \mathbf{q}_G(snk(\alpha)) \times cns(\alpha)$. From Fact 4.5, whenever G' is a connected subgraph of G and $A \in actors(G')$, we have $\mathbf{q}_{G'}(A) \leq \mathbf{q}_G(A)$. Thus, as a loose interdependence algorithm decomposes a strongly connected component into finer and finer components, the amount of delay required for a given edge to be ignorable (within a minimal schedule period) decreases, in general.

In contrast to the top-down approach of outside-in vectorization, Muraoka's inside-out vectorization works by examining the innermost loops first and working outward. If both techniques are fully applied, inside-out vectorization and outside-in vectorization yield the same result. However, the outside-in method is computationally more efficient since a statement that can be vectorized for a series of nested loops is examined once rather than repeatedly for each loop.

6

Joint Code and Data Minimization

This chapter and the next discuss the problem of computing a single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules. Thus, given the model of buffer implementation defined in Section 4.2, we wish to construct a software implementation that minimizes the data memory requirement over all minimum code-size implementations. Even for chain-structured SDF graphs, the number of distinct valid single appearance schedules increases combinatorially with the number of actors [Murt94a], and thus exhaustive evaluation is not, in a general, a feasible means to find the single appearance schedule that minimizes the buffer memory requirement. Section 6.3 develops an efficient dynamic programming algorithm that computes an optimal hierarchy of loops given a lexical ordering of the actors. For well-ordered graphs, where there is only one topological sorting of the vertices, the schedule that results from applying the dynamic programming algorithm is guaranteed to be the optimal one. For graphs that have more than one topological sort, we develop heuristics in Section 6.4 and Chapter 7 for generating suitable topological sorts. These are nested optimally using the dynamic programming algorithm.

6.1 R-Schedules

If Λ is either a schedule loop or a looped schedule, we say that Λ satisfies the **R-condition** if one of the following two conditions holds.

(a) Λ has a single iterand, and this single iterand is an actor, *or*

(b) Λ has exactly two iterands, and these two iterands are schedule loops having coprime iteration counts.

We call a valid single appearance schedule S an **R-schedule** if S satisfies the R-condition, and every schedule loop contained in S satisfies the R-condition.

In [Murt94c] it is shown that in a delayless chain-structured SDF graph, whenever a valid single appearance schedule exists, an R-schedule can be derived whose buffer memory requirement is no greater than that of the original schedule. This result is easily generalized to give the following theorem for arbitrary consistent SDF graphs.

Theorem 6.1: Suppose that $G = (V, E)$ is a consistent SDF graph and S is a valid single appearance schedule for S. Then there exists an R-schedule S_R for S such that $max_tokens(e, S_R) \le max_tokens(e, S)$ for all $e \in E$, and $lexorder(S_R) = lexorder(S)$.

Proof: We prove this theorem by construction. We use the following notation here: given a schedule loop L and a looped schedule S', we define $nonR(S')$ to be the set of schedule loops in S' that do not satisfy the R-condition; $I(L)$ to be the number of iterands of L; and $C(L)$ to be the iteration count of L. Also, we define

$$\hat{I}(S') \equiv \sum_{L' \in nonR(S')} I(L').$$

First observe that from Theorem 4.4, there exists a valid fully reduced schedule S_0 for G such that $max_tokens(e, S_0) \le max_tokens(e, S)$ for all $e \in E$. Now let $L_0 = (nT_1T_2...T_m)$ be an innermost loop in $(1S_0)$ [1] that does not satisfy the R-condition; that is, L_0 does not satisfy the R-condition, but all loops nested in L_0 satisfy the R-condition. If $m = 1$, then since S_0 is fully reduced, $L_0 = (n(1T'))$, where $(1T')$ satisfies the R-condition. Let S^* denote the schedule that results from replacing L_0 with (nT') in $(1S_0)$. Then clearly, S^* is also valid and fully reduced, and S^* generates the same invocation sequence as S_0, so $max_tokens(e, S^*) = max_tokens(e, S_0)$ for all $e \in E$. Also, replacing L_0 with

[1] This is the schedule loop whose iteration count is one and whose body is S_0. Any schedule loop of the form (mS_0) is acceptable for the purposes of this proof, and we have chosen $m = 1$ only for simplicity.

(nT') reduces the number of non-R loops by one, and does not increase the number of iterands of any loop, and thus, $\hat{I}(S^*) < \hat{I}((1S_0))$.

If on the other hand $m \ge 2$, we define $S_a \equiv (1T_1)$ if T_1 is an actor and $S_a \equiv T_1$ if T_1 is a schedule loop. Also, if $T_2, T_3, ..., T_m$ are all schedule loops, we define

$$S_b \equiv \left(\gamma\left(\frac{C(T_2)}{\gamma}B_2\right)\left(\frac{C(T_3)}{\gamma}B_3\right)...\left(\frac{C(T_m)}{\gamma}B_m\right)\right),$$

where $\gamma = gcd(\{C(T_2), C(T_3), ..., C(T_m)\})$, and $B_2, B_3, ..., B_m$ are the bodies of $T_2, T_3, ..., T_m$, respectively; if $T_2, T_3, ..., T_m$ are not all schedule loops, we define $S_b \equiv (1T_2T_3...T_m)$. Let S^* be the schedule that results from replacing L_0 with $L_0' = (nS_aS_b)$ in $(1S_0)$. Now, because S_0 is fully reduced, the iteration counts of S_a and S_b must be coprime. Thus, it is easily verified that S^* is a valid, fully reduced schedule and that L_0' satisfies the R-condition, and with the aid of Theorem 4.2, it is also easily verified that $max_tokens(e, S^*) \le max_tokens(e, S_0)$ for all $e \in E$.

Furthermore, observe that S_a and L_0' satisfy the R-condition, but S_b may or may not satisfy the R-condition, depending on L_0. Thus, replacing L_0 with L_0' either reduces the number of loops that do not satisfy the R-condition by one, or it leaves the number of loops that do not satisfy the R-condition unchanged, and we see that either

$$\hat{I}(S^*) = \hat{I}((1S_0)) - I(L_0), \text{ or } \hat{I}(S^*) = \hat{I}((1S_0)) - I(L_0) + I(S_b).$$

Since $I(S_b) = I(L_0) - 1 < I(L_0)$, we again conclude that $\hat{I}(S^*) < \hat{I}((1S_0))$.

Thus, from $(1S_0)$, we have constructed a valid, fully reduced schedule S^* such that $max_tokens(e, S^*) \le max_tokens(e, S_0) \le max_tokens(e, S)$ for all $e \in E$, and $\hat{I}(S^*) < \hat{I}((1S_0))$. Also, since S^* is derived from S_0 by replacing a single loop that has iteration count n with another loop that has the same iteration count, it is easily verified that S^* is of the form $S^* = (1S_1)$. Clearly, if $\hat{I}((1S_1)) \ne 0$, we can repeat the above process to obtain a valid, fully reduced schedule $(1S_2)$ such that $max_tokens(e_2, (1S_2)) \le max_tokens(e_2, (1S_1))$ for all $e \in E$, and $\hat{I}((1S_2)) < \hat{I}((1S_1))$. Continuing in this manner, we obtain a sequence of valid, fully reduced schedules $((1S_0), (1S_1), (1S_2), (1S_3), ...)$ such that for each S_i in the sequence with $i > 0$,

$$max_tokens(e, (1S_i)) = max_tokens(e, S_i) \le max_tokens(e, S) \text{ for all } e \in E,$$

and $I((1S_i)) < I((1S_{i-1}))$. Since $I((1S_0))$ is finite, we cannot go on generating S_i's indefinitely — eventually, we will arrive at an S_n, $n \geq 0$, such that $\hat{I}((1S_n)) = 0$. Thus, all schedule loops in S_n satisfy the R-condition, and S_n satisfies the R-condition, and we have that S_n is an R-schedule.

From Theorem 4.4 and from the observation that the factoring transformation does not affect the lexical ordering, it is easily verified that none of the transformations applied in deriving S_n from S change the lexical ordering. Thus, $lexorder(S_n) = lexorder(S)$. ■

6.2 The Buffer Memory Lower Bound for Single Appearance Schedules

Given a consistent SDF graph G, there is an efficiently computable upper and lower bound on the buffer memory requirement over all valid single appearance schedules. The lower bound can be derived easily by examining a generic two-actor SDF graph, as shown in Figure 6.1(a). From the balance equations it is easily verified that the repetitions vector for this graph is given by $\mathbf{q}(A, B) = (q/g, p/g)$, where $g \equiv gcd(\{p, q\})$, and that if $d < (pq)/g$, then the only R-schedule for this graph is $S_1 = ((q/g)A)((p/g)B)$. From Theorem 6.1 it follows that if $d < (pq)/g$, then $max_tokens((A, B), S_1) = ((pq)/g + d)$ is a lower bound for the buffer memory requirement of the graph in Figure 6.1(a). Similarly, if $d \geq (pq)/g$, then there are exactly two R-schedules — S_1 and $S_2 = ((p/g)B)((q/g)A)$. Since $max_tokens((A, B), S_2) = d$, we obtain d as a lower bound for the buffer memory requirement. Thus, given a valid single appearance schedule S for Figure 6.1(a), we have that

$$\left(d < \frac{pq}{g}\right) \Rightarrow \left(max_tokens((A, B), S) \geq \left(\frac{pq}{g} + d\right)\right), \text{ and}$$

$$\left(d \geq \frac{pq}{g}\right) \Rightarrow (max_tokens((A, B), S) \geq d). \tag{6.1}$$

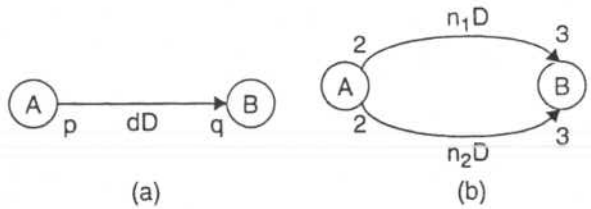


Figure 6.1. Examples used to develop the buffer memory lower bound.

Furthermore, if (A, B) is an edge in a general SDF graph, we know from Fact 4.4 that the projection of a valid schedule S onto $\{A, B\}$, which is a valid schedule for $subgraph(\{A, B\})$, always satisfies

$$max_tokens((A, B), projection(S, \{A, B\})) = max_tokens((A, B), S). \tag{6.2}$$

It follows that the lower bound defined by (6.1) holds whenever (A, B) is an edge in a consistent SDF graph G, S is a valid single appearance schedule for G, $prd((A, B)) = p$, $cns((A, B)) = q$, and $g = gcd(\{p, q\})$. We have motivated the following definition.

Definition 6.1: Given an SDF edge e, we define the **buffer memory lower bound (BMLB)** of e, denoted $BMLB(e)$, by

$$BMLB(e) = \begin{cases} (\eta(e) + del(e)) \text{ if } (del(e) < \eta(e)) \\ (del(e)) \text{ if } (del(e) \geq \eta(e)) \end{cases}, \text{ where}$$

$$\eta(e) = \frac{prd(e)cns(e)}{gcd(\{prd(e), cns(e)\})}$$

If $G = (V, E)$ is an SDF graph, then

$$\left(\sum_{e \in E} BMLB(e)\right)$$

is called the BMLB of G, and a valid single appearance schedule S for G that satisfies $max_tokens(e, S) = BMLB(e)$ for all $e \in E$ is called a **BMLB schedule** for G.

In Figure 6.2, we see that $BMLB((A, B)) = 3$, and $BMLB((B, C)) = 3$. Thus, to implement any single appearance schedule for this graph, at least three memory words will be required to implement the edge (A, B), and at least three words will be required for (B, C). Furthermore, a valid single appearance schedule for Figure 6.2 is a BMLB schedule if and only if its buffer memory requirement equals 6. It is easily verified that only two R-schedules for Figure 6.2 exist — $(3A(2B))(2C)$, and $(3A)(2(3B)C)$; the associated buffer memory require-

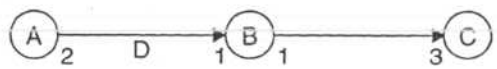


Figure 6.2. An SDF graph that does not have a BMLB schedule.

ments are $3+6 = 9$ and $7+3 = 10$, respectively. Thus, a BMLB schedule does not exist for Figure 6.2.

In contrast, the SDF graph shown in Figure 6.3 has a BMLB schedule. This graph results from simply interchanging the production and consumption parameters of edge (B, C) in Figure 6.2. Here, $\mathbf{q}(A, B, C) = (1, 2, 6)$, the BMLB values for both edges are again identically equal to 3, and $A(2B(3C))$ is a valid single appearance schedule whose buffer memory requirement achieves the sum of these BMLB values.

The following fact is a straightforward extension of our development of the BMLB.

Fact 6.1: Suppose that G is an SDF graph that consists of two vertices A, B and $n \geq 1$ edges $e_1, e_2, \ldots, e_n$ directed from A to B. Then (a) if $del(e_i) \geq \eta(e_i)$ for all $i \in \{1, 2, \ldots, n\}$, then $(\mathbf{q}_G(B)B)(\mathbf{q}_G(A)A)$ is a BMLB schedule for G; (b) otherwise, $(\mathbf{q}_G(A)A)(\mathbf{q}_G(B)B)$ is an optimal schedule — that is, it minimizes the buffer memory requirement over the two valid minimal single appearance schedules — for G, and it is a BMLB schedule if and only if $del(e_i) < \eta(e_i)$ for $1 \leq i \leq n$[2].

For example, in Figure 6.1(b), let e_1 denote the upper edge, and let e_2 denote the lower edge. Then $\eta(e_1) = \eta(e_2) = 6$, and $(2B)(3A)$ is a BMLB schedule if $del(e_1), del(e_2) \geq 6$. Similarly, if $del(e_1), del(e_2) < 6$, then it is easily verified that $(3A)(2B)$ is a BMLB schedule. However, if $del(e_1) < 6$ and $del(e_2) \geq 6$, then $(3A)(2B)$ is optimal, but is not a BMLB schedule since in this case $max_tokens(e_2, (3A)(2B)) = (del(e_2) + 6)$, while $BMLB(e_2) = del(e_2)$.

Fact 6.2: If $G = (V, E)$ is a connected, consistent, acyclic SDF graph, $del(e) < \eta(e)$ for all $e \in E$, and S is a BMLB schedule for the delayless version of G, then S is a BMLB schedule for G.

Proof: Let G' denote the delayless version of G. If S is a BMLB schedule for G', then S is a valid schedule for G that satisfies

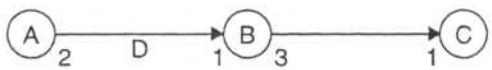


Figure 6.3. An SDF graph that has a BMLB schedule.

2. Note that in this case, $(\mathbf{q}_G(A)A)(\mathbf{q}_G(B)B)$ is the only valid minimal single appearance schedule.

$max_tokens(e, S, G) = max_tokens(e, S, G') + del(e)$ for all $e \in E$. It follows from Definition 6.1 that S is BMLB schedule for G. ■

Fact 6.3: If G is a connected, consistent SDF graph and e is an edge in G, then

$$\eta(e) = \frac{TNSE(e, G)}{\rho_G(src(e), snk(e))}.$$

Recall that $TNSE(e, G)$ is the total number of samples exchanged in one complete cycle of a minimal schedule on edge e in graph G, and $\rho_G(src(e), snk(e))$ is the *gcd* of the repetitions of $src(e)$ and $snk(e)$.

Proof: From the balance equations,

$$\frac{TNSE(e, G)}{\rho_G(src(e), snk(e))} = \frac{\mathbf{q}_G(src(e))prd(e)}{gcd(\{\mathbf{q}_G(src(e)), \mathbf{q}_G(snk(e))\})}$$
$$= \frac{\mathbf{q}_G(src(e))prd(e)}{gcd(\{\mathbf{q}_G(src(e)), \mathbf{q}_G(src(e))(prd(e)/cns(e))\})}.$$

Multiplying the numerator and denominator of this last quotient by $cns(e)$, and recalling that $gcd(ka, kb) = kgcd(a, b)$, we obtain the desired result. ■

We conclude this section by defining an obvious, efficiently computable upper bound for single appearance schedules that have unit blocking factor. Clearly, if $G = (V, E)$ is a connected, consistent SDF graph, and S is a unit blocking factor single appearance schedule for G, we have

$$buffer_memory(S) \leq \sum_{e \in E} (TNSE(e) + del(e)).$$

We refer the RHS of this inequality as the **buffer memory upper bound (BMUB)** for G.

In Figure 6.3, $\mathbf{q}(A, B, C) = (1, 2, 6)$, and the BMUB for this graph is 9.

6.3 Dynamic Programming Post Optimization

In [Murt94c], a dynamic programming algorithm is developed that constructs an optimal schedule for a *well-ordered* SDF graph (a graph that has only one topological sort) in $O(v^3)$ time, where v is the number of actors. An adaptation of this technique is also presented for general, delayless, consistent SDF graphs[3] that computes an **order-optimal** schedule — a single appearance sched-

ule that has minimum buffer memory requirement from among the single appearance schedules that have a given lexical ordering. We refer to this adaptation as **Dynamic Programming Post Optimization (DPPO)** for single appearance schedules. Given a single appearance schedule S, DPPO computes a single appearance schedule that minimizes the buffer memory requirement over all schedules in the set $\{S' | (lexorder(S') = lexorder(S))\}$. Thus, DPPO can be used as a post-optimization to any scheduling technique for delayless, acyclic SDF graphs. This section elaborates on this technique and presents an efficient extension to handle delays and general SDF graphs.

Suppose that G is a connected, consistent, delayless SDF graph, S is valid single appearance schedule for G, $lexorder(S) = (A_1, A_2, \ldots, A_n)$, and S_{oo} is an order optimal schedule for $(G, lexorder(S))$. Assuming that G contains at least two actors, we know from Theorem 6.1 that there is a valid schedule of the form $S_R = (i_L B_L)(i_R B_R)$ such that $buffer_memory(S_R) = buffer_memory(S_{oo})$ and for some $p \in \{1, 2, \ldots, (n-1)\}$, $lexorder(B_L) = (A_1, A_2 \ldots, A_p)$ and $lexorder(B_R) = (A_{p+1}, A_{p+2}, \ldots, A_n)$. Furthermore, from the order-optimality of S_{oo}, clearly, $(i_L B_L)$ and $(i_R B_R)$ must also be order optimal.

From this observation, we can efficiently compute an order-optimal schedule for G if we are given an order optimal schedule $S_{a,b}$ for the subgraph corresponding to each proper subsequence $A_a, A_{a+1}, \ldots, A_b$ of $lexorder(S)$ such that (1). $(b-a) \le (n-2)$ *and* (2). $a = 1$ or $b = n$. Given these schedules, an order-optimal schedule for G can be derived from any value of x, $1 \le x < n$ that minimizes

$$buffer_memory(S_{1,x}) + buffer_memory(S_{x+1,n}) + \sum_{e \in E_s} TNSE(e),$$

where $E_s = \{e | (src(e) \in \{A_1, A_2, \ldots, A_x\}$ **and** $snk(e) \in \{A_{x+1}, A_{x+2}, \ldots, A_n\})\}$ is the set of edges that "cross the split" if the schedule parenthesization is split between A_x and A_{x+1}.

DPPO is based on repeatedly applying this idea in a bottom-up fashion to the given lexical ordering $lexorder(S)$. First, all two actor subsequences (A_1, A_2), (A_2, A_3), $\ldots$, (A_{n-1}, A_n) are examined and the minimum buffer

[3]. Note that for consistent SDF graphs, *delayless* implies *acyclic*, and thus, we are referring here to the class of consistent, acyclic — but not necessarily well-ordered — SDF graphs such that the delay on each edge is zero.

memory requirements for the edges contained in each subsequence is recorded. This information is then used to determine an optimal parenthesization split and the minimum buffer memory requirement for each three actor subsequence (A_i, A_{i+1}, A_{i+2}); the minimum requirements for the two- and three-actor subsequences are used to determine the optimal split and minimum buffer memory requirement for each four actor subsequence; and so on, until an optimal split is derived for the original n-actor sequence $lexorder(S)$. An order-optimal schedule can easily be constructed from a recursive, top-down traversal of the optimal splits [Murt94c].

In the rth iteration of this bottom up approach, we have available the minimum buffer memory requirement $b[p, q]$ for each subsequence $(A_p, A_{p+1}, \ldots, A_q)$ that has less than or equal to r members. To compute the minimum buffer memory requirement $b[i, j]$ associated with an $r+1$-actor subchain $(A_i, A_{i+1}, \ldots, A_j)$, we determine a value of $k \in \{i, i+1, \ldots, j-1\}$ that minimizes

$$b[i, k] + b[k+1, j] + c_{i,j}[k], \tag{6.3}$$

where $b[x, x] = 0$ for all x and $c_{i,j}[k]$, the memory cost at the split if we split the subsequence between A_k and A_{k+1} is given by (see Theorem 4.3).

$$c_{i,j}[k] = \frac{\sum_{e \in E_s} TNSE(e)}{gcd(\mathbf{q}_G(A_x) | (i \le x \le j))}, \tag{6.4}$$

where

$$E_s = \{e | (src(e) \in \{A_i, A_{i+1}, \ldots, A_k\} \text{ and } snk(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\} \tag{6.5}$$

is the set of edges that cross the split.

This technique can be extended to handle graphs that are not necessarily delayless, although a few additional considerations arise. We refer to our extension as **Generalized DPPO (GDPPO)**. First, if delays are present, then Fact 4.2 does not apply, and $lexorder(S)$, the lexical ordering of the input schedule, is not necessarily a topological sort. As a consequence, generally not all parenthesizations of the input schedule will be valid. For example, suppose that we are given the valid schedule $S = (6A)(5(2C)(3B))$ for Figure 6.4. Then $lexorder(S) = (A, C, B)$ clearly is not a topological sort, and it is easily verified that the schedule that corresponds to splitting the outermost parenthesization

between C and B — $(2(3A)(5C))(15B)$ — is not a valid schedule since there is not sufficient delay on the edge (B, C) to fire 10 invocations of C before a single invocation of B.

Thus, we see that when delays are present, the set E_s defined in (6.5) no longer generally gives all of the edges that cross the parenthesization split. We must also examine the set of *back edges*

$$E_b = \{e | (snk(e) \in \{A_i, A_{i+1}, \ldots, A_k\} \text{ and } src(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\}. \quad (6.6)$$

Each $e \in E_b$ must satisfy

$$del(e) \geq \frac{TNSE(e)}{gcd(\mathbf{q}_G(A_x) | (i \leq x \leq j))}, \quad (6.7)$$

otherwise the given parenthesization split will give a schedule that is not valid. To take into account any nonzero delays on members in E_s, and the memory cost of each of the back edges, the cost expression of (6.3) for the given split gets replaced with

$$b[i,k] + b[k+1,j] + \frac{\sum_{e \in E_s} TNSE(e)}{gcd(\mathbf{q}_G(A_x) | (i \leq x \leq j))} + \sum_{e \in E_s} del(e) + \sum_{e \in E_b} del(e). \quad (6.8)$$

Expression (6.8) gives the cost of splitting the subsequence $(A_i, A_{i+1}, \ldots, A_j)$ between A_k and A_{k+1} assuming that the subsequence $(A_i, A_{i+1}, \ldots, A_k)$ precedes $(A_{k+1}, A_{k+2}, \ldots, A_j)$ in the lexical order of the schedule that will be implemented. However, if (6.7) is satisfied *for all* "forward edges" $e \in E_s$, it may be advantageous to interchange the lexical order of $(A_i, A_{i+1}, \ldots, A_k)$ and $(A_{k+1}, A_{k+2}, \ldots, A_j)$. Such a reversal will be advantageous whenever the *reverse split cost* defined by

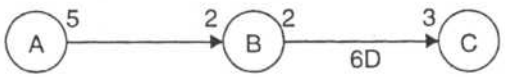


Figure 6.4. An SDF graph used to illustrate GDPPO applied to SDF graphs that have nonzero delay on one or more edges. Here $\mathbf{q}(A, B, C) = (6, 15, 10)$.

$$b[i,k] + b[k+1,j] + \frac{\sum_{e \in E_b} TNSE(e)}{gcd(\mathbf{q}_G(A_x) | (i \leq x \leq j))} + \sum_{e \in E_b} del(e) + \sum_{e \in E_s} del(e) \quad (6.9)$$

is less than the *forward split cost* computed from (6.8).

The possibility for reverse splits introduces a fundamental difference between GDPPO and DPPO: if one or more reverse splits are found to be advantageous, then GDPPO does not preserve the lexical ordering of the original schedule. This is not a problem since in such cases the result computed by GDPPO will necessarily have a buffer memory requirement that is less than that of an order-optimal schedule for $lexorder(S)$. On the contrary, it suggests that GDPPO may be applied multiple times in succession to yield more benefit than a single application — that is, GDPPO can in general be applied iteratively, where the iterative application terminates when the schedule produced by GDPPO produces no improvement over the schedule computed in the previous iteration.

Figure 6.5 shows an example where multiple applications of GDPPO is beneficial. Here $\mathbf{q}(A, B, C) = (2, 1, 2)$, and the initial schedule is $S = (2A)B(2C)$, so the initial lexical ordering is (A, B, C). Upon application of GDPPO, the minimum cost for the subsequence (A, B) is found to be 2, and the minimum cost for the subsequence (B, C) is found to occur with a reverse split that has a cost of 2. The minimum cost for the "top-level" subsequence (A, B, C) is taken as the minimum cost over the cost if the parenthesization is split between A and B, which is equal to $0 + 2 + 5 + 0 = 7$ from (6.9), and the minimum cost if the split occurs between B and C, which is $2 + 0 + 7 + 0 = 9$. Thus, the former split is taken, and the result of applying GDPPO *once* to S is the schedule $S_1 = (2A)(1(2C)(1B))$, which has a buffer memory requirement of 7, and a lexical ordering that is different from that of S.

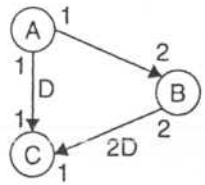


Figure 6.5. An illustration of iterative application of GDPPO.

Since $lexorder(S_1) \neq lexorder(S)$, it is conceivable that applying GDPPO to S_1 can further reduce the buffer memory requirement. Applying GDPPO to S_1, we generate a minimum cost of 1 — which corresponds to another reverse split — for (A, C), and we generate a minimum (reverse split) cost of 2 for (C, B). Thus, we see that splitting (A, C, B) between A and C gives a cost of $0+2+5+0 = 7$, while splitting between C and B gives a cost of $1+0+2+2 = 5$. The result of GDPPO is thus the schedule $S_2 = (2CA)B$, and a buffer memory requirement of 5. It is easily verified that application of GDPPO to S_2 yields no further improvement, and thus iterative application of GDPPO terminates after three iterations.

Although the iterative application of GDPPO is conceptually interesting, we have found that for all of the practical SDF graphs that we have applied it to, termination occurred after only 2 iterations, which means that no further improvement was ever generated by a second application of GDPPO. This suggests that when compile-time efficiency is a significant issue, it may be preferable to bypass iterative application of GDPPO, and immediately accept the schedule produced by the first application.

GDPPO can be implemented efficiently by updating forward and reverse costs incrementally. If we are examining the splits of the subsequence $(A_i, A_{i+1}, \ldots, A_j)$, and we have computed the forward and reverse split costs F_k and R_k associated with the split between A_k and A_{k+1}, $i \leq k < (j-1)$, then the splits costs F_{k+1} and R_{k+1} associated with the split between A_{k+1} and A_{k+2} can easily be derived by examining the output and input edges of A_{k+1}. To ensure that we ignore reverse splits (forward splits) that fail to satisfy (6.7) for all $e \in E_s$ ($e \in E_b$) a cost of

$$M \equiv \left(1 + \sum_{e \in E} (TNSE(e) + del(e))\right)$$

is added to the reverse (forward) split cost for any input edge (output edge) e of A_{k+1} whose source (sink) is a member of $(A_{k+2}, A_{k+3}, \ldots, A_j)$, and that does not satisfy (6.7). Similarly, for each output (input) edge e of A_{k+1} whose sink (source) is contained in $(A_i, A_{i+1}, \ldots, A_k)$, and that does not satisfy (6.7), M is subtracted from R_{k+1} (F_{k+1}) since such an edge no longer prevents the split from being valid. Choosing M so large has the effect of "invalidating" any cost C_M that has M added to it (without a corresponding subtraction) since any minimal valid schedule has a buffer memory requirement less than M, and thus, any valid split will be chosen over a split that has cost C_M.

If forward and reverse costs are updated in this incremental fashion, then GDPPO attains a time complexity of $O(n_v^3)$ where n_v is the number of actors, if we can assume that the number of input and output edges of each actor is always bounded by some constant α. In the absence of such a bound, GDPPO has time complexity that is $O(n_e n_v^3)$, where n_e is the number of edges in the input graph.

6.3.1. Well-ordered Graph: Sample Rate Conversion Example

For graphs that are well-ordered (that is, have only one topological sort), GDPPO will give the single appearance schedule that has the least buffer memory requirement. A practical example from this class of graphs is a sample-rate conversion system. Digital audio tape (DAT) technology operates at a sampling rate of 48 kHz, while compact disk (CD) players operate at a sampling rate of 44.1 kHz. Interfacing the two, for example, to record a CD onto a digital tape, requires a sample rate conversion.

The naive way to do this is shown in Figure 6.6(a). It is more efficient to perform the rate change in stages. Rate conversion ratios are chosen by examining the prime factors of the two sampling rates. The prime factors of 44,100 and 48,000 are $2^2 3^2 5^2 7^2$ and $2^7 3^1 5^3$, respectively. Thus, the ratio 44,100:48,000 is $3^1 7^2 : 2^5 5^1$, or 147:160. One way to perform this conversion in four stages is 2:1, 4:3, 4:7, and 5:7. Figure 6.6(b) shows the multistage implementation. Explicit upsamplers and downsamplers are omitted, and it is assumed that the FIR filters are general polyphase filters [Vaid93].

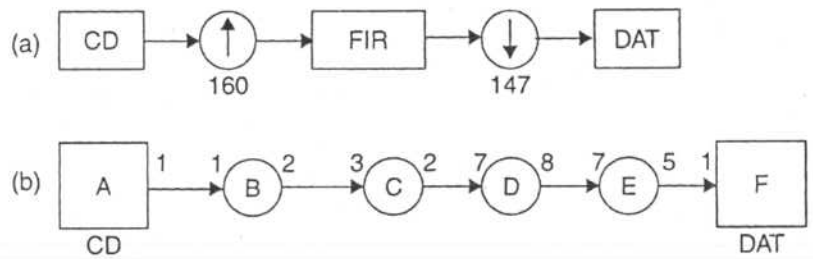


Figure 6.6. (a). CD to DAT sample rate change system. (b). Multi-stage implementation of a CD to DAT sample rate system.

Here $\mathbf{q}(A, B, C, D, E, F) = (147, 147, 98, 28, 32, 160)^{\cdot}$; the optimal looped schedule given by GDPPO is $(7(7(3AB)(2C))(4D))(32E(5F))$; and the associated buffer memory requirement is 264 . In contrast, the alternative schedule $(147A)(147B)(98C)(28D)(32E)(160F)$ has a buffer memory requirement of 1021 if a separate buffer is used for each edge, and a buffer-memory requirement of 294 if one shared buffer is used.

We note briefly that nested-schedules have a lower latency than flat single appearance schedules. The **latency** is defined to be the time at which the sink vertex fires for the first time in the schedule. In a flat schedule $(\mathbf{q}(N_1)N_1)(\mathbf{q}(N_2)N_2)\ldots(\mathbf{q}(N_K)N_K)$, the latency is given by

$$(\mathbf{q}(N_1) - 1)T + E_1 + \sum_{i=2}^{K-1} \mathbf{q}(N_i)E_i,$$

where T is the sample period of the source actor, and E_i is the execution time of actor N_i. All these times are assumed to be in number of instruction cycles of the processor. A nested-schedule will usually have a latency less than this because if the sink actor is part of a nested loop body, then all of the invocations of actors upstream do not have to occur before the sink actor fires for the first time.

It can be verified that the latency of the optimally nested schedule is given by

$$146T + E_A + E_B + 2E_C + 4E_D + E_E,$$

as opposed to

$$146T + E_A + 147E_B + 98E_C + 28E_D + 32E_E$$

for the flat schedule. If we take $T = 500$ (for example, a 22.05Mhz chip has 22.05Mhz/44.1khz = 500 instruction cycles in one sample period of the CD actor), and $E_A = 10, E_B = E_C = E_D = E_E = 100$, then the two latencies are 73810 and 103510 instruction cycles; the nested schedule has 29% less latency.

One more advantage that a nested schedule can have over the flat schedule with shared buffering is in the amount of **input buffering** required. Some DSP chips have a feature where a dedicated I/O manager can write incoming samples to a buffer in on-chip memory, the size of which can be programmed by the user. If the single appearance schedule spans more than one sample period, then input buffering is a useful feature since it avoids the need for interrupts. Chips that have

input buffering include the Analog Devices ADSP 2100. If we compute the amount of input buffering required by the flat schedule, we find that it is $((147 + 98 + 28 + 32)100 + 160 \times 10)/500 \cong 65$, whereas for the optimally nested schedule, it is given by
$(100 + 200 + 400 + 3200 + 1600)/500 \cong 11$.

GDPPO gives a post-optimization that can be appended to any scheduler for general SDF graphs that constructs single appearance schedules. Applying GDPPO to a single appearance schedule S yields a schedule that has a buffer memory requirement that is less than or equal to the buffer memory requirement of every valid single appearance schedule that has the same lexical ordering as S. The remainder of the book discusses two heuristics for constructing single appearance schedules. Section 6.5 gives a practical example of a non-well-ordered graph where GDPPO improves the schedules obtained by these two heuristics significantly. An experimental study that compares these heuristics — with their schedules post-processed by GDPPO — against each other and against randomly generated schedules that are post-processed by GDPPO is presented in Chapter 8.

The basic concept behind DPPO can be applied to derive an algorithm that computes an optimally compact looped schedule (minimum code size) for an arbitrary sequence of actor firings. For example, consider the SDF graph in Fig. 6.7, and suppose that we are given the valid firing sequence[4] $\sigma = ABABCBC$ (this firing sequence minimizes the buffer memory requirement over all valid schedules). If the code size cost (number of program memory words required) for a code block for A is greater than the code size cost for C, then the optimally compact looped schedule for σ is $(2AB)CBC$, whereas $ABA(2BC)$ is optimal if C has a greater code size cost than A. The details of this modified dynamic programming technique can be found in [Bhat95].

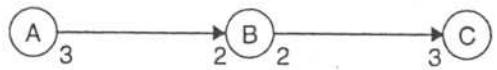


Figure 6.7. An SDF graph used to illustrate the problem of finding an optimally compact loop structure for an arbitrary firing sequence.

4. By a *firing sequence*, we simply mean a schedule that contains no loops.

6.4 Recursive Partitioning by Minimum Cuts (RPMC)

The number of topological sorts in an acyclic graph can be an exponential function of the size of the graph; for example, a complete bipartite graph with $2n$ vertices has $(n!)^2$ possible topological sorts. Each topological sort gives a valid flat single appearance schedule. An optimal reparenthesization of this schedule is then computed by applying the dynamic programming algorithm. The problem is therefore to determine the topological sort that will give the lowest buffer memory requirement when nested optimally. For example, the graph in Figure 6.8 shows a bipartite graph with 4 vertices. The repetitions vector for the graph is given by $(12, 36, 9, 16)^T$, and there are 4 possible topological sorts for the graph. The flat schedule corresponding to the topological sort $ABCD$ is given by $(12A)(36B)(9C)(16D)$. This can be parenthesized as $(3(4A)(3(4B)C))(16D)$, and this schedule has a buffer memory requirement of 208. The flat schedule corresponding to the topological sort $ABDC$, when parenthesized optimally, gives the schedule $(4(3A)(9B)(4D))(9C)$, with a buffer memory requirement of 120.

A heuristic solution for this problem can be formulated as follows: find the *cut* (a partition of the set of actors) of the graph across which the minimum amount of data is transferred and schedule the resulting halves recursively. The cut that is produced must have the property that all edges that cross the cut have the same direction. This is to ensure that all the vertices on the left side of the partition can be scheduled before scheduling any on the right side. In addition, we would like to impose the constraint that the partition that results be fairly evenly sized. This is to increase the possibility of having gcds that are greater than unity for the repetitions of the vertices in the subsets produced by the partition, thus reducing the buffer memory requirement. To see that having gcds greater than one for the subsets produced is beneficial to memory reduction, consider Figure 6.8. If

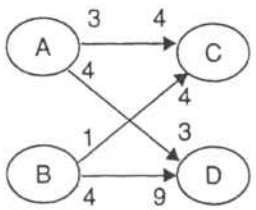


Figure 6.8. A bipartite SDF graph to illustrate the different buffer memory requirements possible with different topological sorts.

the partition that had actor B on one side of the cut and actors A, C, D on the other side of the cut were formed, then we would get the loop bodies $(36B)$ and $((12A)(9C)(16D))$ and would not immediately see a reduction in buffering requirements since the repetitions of A, C, D are co-prime. However, a partition with A, B, C on the same side of the cut immediately gives us a reduction since the schedule body $((12A)(36B)(9C))$ can be factored as $(3(4A)(12B)(3C))$ (see Theorem 4.2), and this reduces the memory for the subgraph consisting of actors A, B, C. In general, by constraining the sizes of the partition, the probability of being able to factor schedule bodies so that a reduction in memory is obtained in each stage of the recursion is increased. Needless to say, this is a greedy approach which is likely to fail sometimes but has proved to be a good rule of thumb for most instances.

Suppose that G is an SDF graph, and let $V = actors(G)$ and $E = edges(G)$. A **cut** of G is a partition of the vertex set V into two disjoint sets V_L and V_R. Define $G_L = subgraph(V_L)$ and $G_R = subgraph(V_R)$ to be the subgraphs produced by the cut. The cut is **legal** if for all edges e *crossing* the cut (that is all edges that are not contained in $subgraph(V_L)$ nor $subgraph(V_R)$), we have $src(e) \in V_L$ and $snk(e) \in V_R$. Given a *bounding constant* $K \le |V|$, the cut results in bounded sets if it satisfies

$$|V_R| \le K, \; |V_L| \le K. \tag{6.10}$$

The weight of an edge e is defined as

$$w(e) = \mathbf{q}_G(src(e)) \times prd(e). \tag{6.11}$$

The weight of the cut is the total weight of all the edges crossing the cut. The problem then is to find the minimum weight legal cut into bounded sets for the graph with the weights defined as in (6.11). Since the related problem of finding a minimum cut (not necessarily legal) into bounded sets is NP-complete [Gare79], and the problem of finding an acyclic partition of a graph is NP-complete [Gare78], we believe this problem to be NP-complete as well even though we have not discovered a proof. Kernighan and Lin [Kern70] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [Corm90] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded. The graph in Figure 6.9, where a weight on an edge denotes the capacity of that edge, illustrates this. The maximum flow into vertex t is seen to

be 3 (1 unit of flow along the path $sBCt$, 1 unit along $sADt$ and 1 unit along $sBDt$) and this corresponds to the cut where $V_L = \{s, B, C\}$ and $V_R = \{A, D, t\}$. The value of the cut is given by $1 + 1 + 1 = 3$ (note that the definition of the value of a cut in network flow theory is defined as sum of the capacities of the edges crossing the cut in the s to t direction only) but the cut is not legal because of the reverse edge from A to C. Indeed, the minimum weight legal cut for this graph has a value of 11, corresponding to the cut where $V_L = \{s\}$.

Therefore, a heuristic solution for finding legal minimum cuts into bounded sets is given. The heuristic is to examine the set of cuts produced by taking a vertex and all of its *descendants* as the vertex set V_R and the set of cuts produced by taking a vertex and all of its *ancestors* as the set V_L. For each such cut, an optimization step is applied that attempts to improve the cost of the cut.

Consider a cut produced by setting $V_L = ancs(v), V_R = V \setminus V_L$ for some vertex v. Consider the set $T_R(v)$ of independent, *boundary* vertices of v in V_R. A **boundary vertex** in V_R is a vertex that is not the predecessor of any other vertex in V_R. Following Kernighan and Lin [Kern70], for each of these vertices, the cost difference that results if the vertex is moved into V_L can be computed efficiently. This cost difference for a vertex a in $T_R(v)$ is defined to be the difference between the total weight of all the edges out of a and the total weight of all edges into a. We then move those vertices across that reduce the cost. We apply this optimization step for all cuts of the form $ancs(v)$ and $desc(v)$ for each vertex v in the graph and take the best one as the minimum cut. The algorithm is shown in Figure 6.10. Since a greedy strategy is being used to move vertices across, and only the boundary vertices are considered, examples can be constructed where the heuristic will not give optimal cuts. Since there are $|V|$ verti-

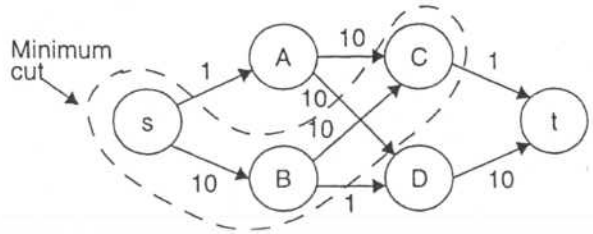


Figure 6.9. The min-cut given by the max-flow-min-cut theorem is not equal to the min-legal cut for this graph.

procedure MinimumLegalCutIntoBoundedSets
input: weighted directed graph $G = (V, E)$, and a bound b. **output**: V_R, V_L.
for each $u \in V$

/* Start with descendents */

$S = desc(u)$, $\bar{S} = V \backslash S$
$cutVal = cut(\bar{S}, S)$
$T_L(u) \leftarrow independent(u) \cap boundary(\bar{S})$
for each $a \in T_L(u)$

$E(a) = \sum_{x \in S} w(a, x)$, $I(a) = \sum_{x \in \bar{S}} w(x, a)$

$D(a) = I(a) - E(a)$ /* Cost difference if this vertex is moved over
end for
$[D, Idx] \leftarrow sort(D)$
$k \leftarrow 1$
while $(|S| < b \ \& \ D(k) < 0 \ \& \ k < |T_L(u)|)$
$S \leftarrow S \cup \{Idx(k)\}$
$\bar{S} \leftarrow \bar{S} \setminus \{Idx(k)\}$
$cutVal \leftarrow cutVal + D(k)$
$k \leftarrow k + 1$
end while
$minCutVal \leftarrow min(minCutVal, cutVal)$
if $(mincutVal \equiv cutVal)$, $V_L \leftarrow \bar{S}$, $V_R \leftarrow S$, **end if**

/* Start with ancestors */

$P = ancs(u)$, $\bar{P} = V \setminus P$

/* Carry out the same type of steps as above to determine the partition for the set starting from the ancestors */

end for
/* $minCutVal, V_L, V_R$ correspond to the minimum legal cut. */

Figure 6.10. Procedure for computing legal minimum cuts.

ces in the graph, $2|V|$ cuts are examined. Moreover, the cut produced will have bounded sets since cuts that produce unbounded sets are discarded. For example, one of the cuts examined by the heuristic for the graph in Figure 6.9, with bounding constant $K = |V| - 1$, is $ancs(A) = \{s, A\}$. This cut has a value of 30. The set of independent, boundary vertices of A in V_R is $\{B\}$, and the cost difference for B is given by $11 - 10 = 1$. Hence, B will not be moved over. The cut produced by considering $ancs(C) = \{s, A, B, C\}$ has a value of 12. The cost difference for the independent vertex D is given by $10 - 11 = -1$; hence, D is moved into V_L to yield a cut of value 11, and thus, in this example, the heuristic finds the minimum weight legal cut.

Delays on edges are handled as follows. If the number of delays on some edge e satisfies

$$del(e) \geq TNSE(e), \tag{6.12}$$

then the size of the buffer on this edge need not be any greater than $del(e)$. However, if e crosses the cut, then the size of the buffer will become $del(e) + TNSE(e)$. Hence, an edge that satisfies equation 6.12, is *tagged*; a tagged edge does not affect the legality of the cut (in other words, the heuristic ignores tagged edges when it constructs the legal cut) but affects the cost of the cut: if a tagged edge crosses the cut in the reverse direction, the cost of the edge is given by $del(e)$, and if the tagged edge crosses the cut in the forward direction, the cost is given by $del(e) + TNSE(e)$. This will discourage the heuristic from choosing partitions where tagged edges cross the cut in the forward direction.

The running time of the heuristic for computing the legal minimum cut into bounded sets can be determined as follows. Computing the descendents or ancestors of a vertex can be done by using breadth-first-search; this takes time $\Theta(|V| + |E|)$. The breadth-first-search will also give us the independent vertices in the complement set. Finding and computing the cost difference for each of the boundary vertices in the set of independent vertices takes at most $O(|E|)$ steps. Sorting the cost differences takes $O(|V| \bullet \log(|V|))$ steps, and moving the vertices that reduce the cost takes $O(|V|)$ time. Since a cut is determined for every vertex twice, the total running time is $O(|V||E| + |V|^2 \bullet \log(|V|))$.

The heuristic for generating a schedule for the acyclic graph now proceeds by partitioning the graph by computing the legal minimum cut and forming the schedule body $(r_L S_L)(r_R S_R)$ where $r_L = gcd(\{\mathbf{q}(v) | v \in V_L\})$, $r_R = gcd(\{\mathbf{q}(v) | v \in V_R\})$ and S_L, S_R are schedule bodies for G_L and G_R

respectively. The schedule bodies S_L, S_R are obtained recursively by partitioning G_L and G_R. Once the entire schedule body has been constructed, the dynamic programming algorithm is run to re-parenthesize the schedule to possibly give a better nesting. Letting $n = |V|$, the running time for this heuristic can be determined by solving the recurrence

$$T(n) = T(n-k) + T(k) + O(n|E| + n^2 \bullet \log(n)),$$

where $k = |V_L|$ and $n - k = |V_R|$. If the bound K in (6.10) is chosen to be a constant factor of the graph size, for example, 3/4, then it can be shown easily that $T(n) = O(|V||E| + |V|^2 \bullet \log(|V|))$. If the size of the sets is not bounded to be a constant factor of the graph size, then the worst case running time is $O(|V|^2|E| + |V|^3 \bullet \log(|V|))$. The reparenthesizing step that is run at the end uses the dynamic programming algorithm and requires $O(|V|^3)$ running time. Thus the overall running time is given by $O(|V|^3)$ assuming that K is a constant factor of the graph size.

6.5 Non-uniform Filterbank Example

Figure 6.11 shows the SDF graph abstraction of a non-uniform, near-perfect reconstruction filterbank [Naye93]. The lowpass filters (vertices c, f etc.) retain 2/3 of the spectrum while the highpass filters (vertices d, g etc.) retain 1/3 (instead of the customary 1/2,1/2 for the octave QMF). Rate changes in the graph are annotated wherever the number produced or consumed is different from unity. The repetitions vector of this graph is given by

$$\mathbf{q}(a, \ldots, A) = [27, 27, 9, 9, 18, 6, 6, 9, 12, 6, 9, 4, 4, 6, 8, 4, 4, 4, 12, 6, 6, 9, 18, 9, 27, 27, 27]$$

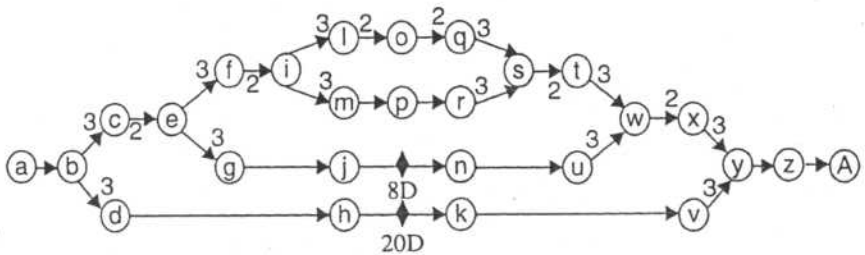


Figure 6.11. SDF graph for a non-uniform filterbank. The highpass channel retains 1/3 of the spectrum and the lowpass channel retains 2/3 of the spectrum.

The single appearance schedule obtained by the APGAN scheduling heuristic (discussed in the next chapter) on this system is

$$(3(3k(3ab)cd(2e)h)(2nugj)(2f(2i)))(4l(2o)q)(4mpr)(3(2(2s)t)(3(2w)xv(3yzA))),$$

and the resulting buffer memory requirement is 153. After post-processing the schedule with GDPPO, we obtain the schedule

$$(3(3(3ab)cd(2e)kh)(2gnufj))(4(3i)l(2o)qmpr(3s))(3(2t)(3(2w)xv(3yzA))),$$

which has a buffer memory requirement of 137 — a 10.5% improvement. Notice that in this example, GDPPO has changed the lexical ordering, and thus, one or more reverse splits were found to be beneficial.

The schedule returned by the RPMC scheduling heuristic has a buffer memory requirement of 131, which is lower than that obtained by the combination of APGAN and GDPPO. However, GDPPO is able to improve the schedule obtained by RPMC even further. The result computed by GDPPO when applied to the schedule derived by RPMC is

$$(3(3(3ab)dc)(2(3e)fgnj)(3kh))(4(3i)mpl(2o)qr(3s))(3(2tu(3w))(3xv(3yzA))),$$

which has a buffer memory requirement of 128.

For the same graph, if the delays are removed from the edges and instead implemented as vertices, then the RPMC heuristic obtains a schedule with a buffering cost of 100; the worst case flat schedule (for any topological sort) would have a buffering cost of 438. The best of the various PGAN heuristics (discussed in Chapter 7) found a schedule of cost 117. This shows that RPMC can be effective on graphs with irregular rate-changes in practice. More extensive experimental results on several other practical examples, and numerous random examples, are reported in Chapter 8.

7

Pairwise Grouping of Adjacent Nodes

In the original *Pairwise Grouping of Adjacent Nodes (PGAN)* technique for joint code and data minimization, a cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step [Bhat93]. At each clustering step, a pair of adjacent actors is chosen that maximizes $\rho(\{A, B\})$, the repetition count of the adjacent pair, over all clusterable adjacent pairs $\{A, B\}$. Recall from Section 4.3 that $\rho(Z)$ can be viewed as the number of times a minimal periodic schedule for the subset of actors Z is invoked in the given SDF graph, and thus, we see that the PGAN technique repeatedly clusters adjacent pairs whose associated subgraphs are invoked most frequently in a valid schedule.

The PGAN technique verifies whether or not an adjacent pair is clusterable by checking whether or not its consolidation introduces a cycle in the Acyclic Precedence Graph (APG). It is shown that this check can be performed quickly by applying a *reachability matrix*, which indicates for any two APG vertices x and y whether or not there is a path from x to y.

Unfortunately, the cost to compute and store the reachability matrix can be prohibitively high for multirate applications that involve large changes in sample rate. Since the number of vertices in the APG of an SDF graph (V, E) is $J \times I$, where J is the desired blocking factor, I is defined as

$$I \equiv \sum_{X \in V} \mathbf{q}(X),$$

and the number of entries in the reachability matrix is quadratic in the number of APG vertices, it is easily seen that the time and space required to maintain the APG can grow exponentially with the number of actors in the given SDF graph. Although this is not a problem for the large class of practical SDF graphs for which I is not much larger than the number of elements in V, practical examples can easily be constructed where the technique consumes enormous amounts of resources relative to the size of the input SDF graph. For example, for the 6-vertex SDF representation of a multi-stage sample-rate conversion system between a compact disc player and a digital audio tape player discussed in Section 6.3.1., $I > 600$, which means that over 360,000 units of storage are required to implement the reachability matrix for this 6-actor SDF graph.

Since a large proportion of DSP applications that are amenable to the SDF model can be represented as acyclic SDF graphs, a simple adaptation of PGAN has been proposed for acyclic graphs that maintains the cluster hierarchy and reachability matrix directly on the input SDF graph rather than on the APG, and thus allows us to efficiently exploit the advantages of the bottom-up clustering approach of the original PGAN technique. We refer to this adaptation of PGAN as **Acyclic PGAN (APGAN)**. APGAN is exactly the original PGAN technique specified in [Bhat93] with the exception that the input SDF graph is assumed to be acyclic, and the cluster hierarchy and reachability matrix are maintained for the input SDF graph rather than for the APG.

In an acyclic SDF graph G, it is easily verified that if a subset Z of actors is not clusterable then $clust(Z, G, \Omega)$ contains a cycle. Whether Z introduces a cycle is easily checked given a reachability matrix for G by examining each successor of a member of Z: $clust(Z, G, \Omega)$ contains a cycle if and only if there is an $X \notin Z$ such that X is a successor of some member of Z, *and* there is a path from X to some member of Z.

Since the existence of a cycle in $clust(Z, G, \Omega)$ is only a necessary — but not sufficient — condition for Z not to be clusterable, the clusterability test that is applied in APGAN is not *exact*; it must be viewed as a conservative test. It is even inexact if we restrict ourselves to single appearance schedules. That is, it is possible for $clust(Z, G, \Omega)$ to contain a cycle, and still have a valid single appearance schedule. A simple example is shown in Figure 7.1. Here, the BMLB schedule $C(2AB)$ results if we first cluster $\{A, B\}$. However in APGAN, clustering $\{A, B\}$ is not permitted since the resulting graph contains a cycle. Instead, APGAN generates the schedule $(2A)C(2B)$ or the schedule $C(2A)(2B)$, neither

of which is a BMLB schedule. Thus, in this example, we see that the inexact clusterization test prevents us from obtaining an optimal schedule.

In exchange for some degree of suboptimality in certain examples, the clusterization test attains a large computational savings over the exact test based on the reachability matrix of the APG, and this is the main reason for adopting it.

Figure 7.2 illustrates the operation of APGAN. Figure 7.2(a) shows the input SDF graph. Here $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and for $i = 1, 2, 3, 4$, Ω_i represents the ith hierarchical actor instantiated by APGAN. Each edge corresponds to a different adjacent pair; the repetition counts of the adjacent pairs are given by

$$\rho(\{A, B\}) = \rho(\{A, C\}) = \rho(\{B, C\}) = 2,$$

and

$$\rho(\{C, D\}) = \rho(\{E, D\}) = \rho(\{B, E\}) = 1.$$

Thus, APGAN will select the one of the three adjacent pairs $\{A, B\}$, $\{A, C\}$, or $\{B, C\}$ for its first clusterization step. Examination of the reachability matrix yields that $\{A, C\}$ introduces a cycle due to the path $((A, B), (B, C))$, while the other two adjacent pairs do not introduce cycles. Thus, APGAN chooses arbitrarily between $\{A, B\}$ and $\{B, C\}$ as the first adjacent pair to cluster.

Figure 7.2(b) shows the graph that results from clustering $\{A, B\}$ into the hierarchical actor Ω_1. In this graph, $\mathbf{q}(\Omega_1, C, D, E) = (2, 4, 5, 1)$, and it is easily verified that $\{\Omega_1, C\}$ uniquely maximizes ρ over all adjacent pairs. Since $\{\Omega_1, C\}$ does not introduce a cycle, APGAN selects this adjacent pair for its second clusterization step. Figure 7.2(c) shows the resulting graph.

In Figure 7.2(c), we have $\mathbf{q}(\Omega_2, D, E) = (2, 5, 1)$, and thus all three adjacent pairs have $\rho = 1$. Among these, clearly, only $\{\Omega_2, E\}$ and $\{E, D\}$ do not

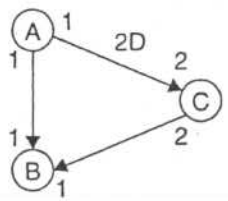


Figure 7.1. An example of how a clusterization operation that introduces a cycle can lead to a BMLB schedule. Here $\mathbf{q}(A, B, C) = (2, 2, 1)$.

introduce cycles, so APGAN arbitrarily selects among these two to determine the third clusterization pair. Figure 7.2(d) shows the graph that results when $\{E, D\}$ is chosen. This graph contains only one adjacent pair $\{\Omega_2, \Omega_3\}$, and APGAN will consolidate this pair in its final clusterization step to obtain the single-vertex graph in Figure 7.2(e).

Figures 7.2(a-e) specify the sequence of clusterizations performed by APGAN when applied to the graph of Figure 7.2(a). A more compact representation of this sequence is shown in Figure 7.3. A valid single appearance schedule for Figure 7.2(a) can easily be constructed by recursively traversing the hierarchy induced by this sequence. We start by constructing a schedule for the top-level subgraph, the subgraph corresponding to Ω_4. The subgraph G_i corresponding to each Ω_i consists of only two actors X_i and Y_i, such that all edges in G_i are directed from X_i to Y_i. Thus, from Fact 6.1, it is clear how an optimal schedule can easily be constructed for the subgraph corresponding to each Ω_i: if each edge

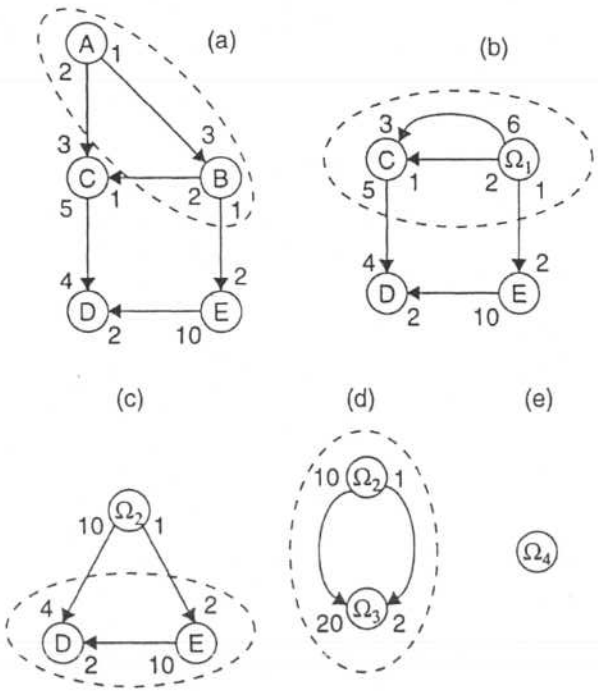


Figure 7.2. An illustration of APGAN.

e in G_i satisfies $del(e) \geq \eta(e)$, then we construct the schedule $(\mathbf{q}_{G_i}(Y_i)Y_i)(\mathbf{q}_{G_i}(X_i)X_i)$, and otherwise we construct $(\mathbf{q}_{G_i}(X_i)X_i)(\mathbf{q}_{G_i}(Y_i)Y_i)$. In Figure 7.2, This yields the "top-level" schedule $(2\Omega_2)\Omega_3$ (we suppress loops that have an iteration count of one) for the subgraph corresponding to Ω_4.

Next, we recursively descend one level in the cluster hierarchy to the subgraph corresponding to Ω_3, and we obtain the schedule $(5D)E$. Since this subgraph contains no hierarchical actors, $(5D)E$ is immediately returned as the "flattened" schedule for the subgraph corresponding to Ω_3. This flattened schedule then replaces its corresponding hierarchical actor in the top-level schedule, and the top-level schedule becomes $(2\Omega_2)(5D)E$.

Next, descending to Ω_2, we construct the schedule $\Omega_1(2C)$ for the corresponding subgraph. We then examine the subgraph corresponding to Ω_1 to obtain the schedule $(3A)B$. Substituting this for Ω_1, the schedule for the subgraph corresponding to Ω_2 becomes $(3A)B(2C)$. Finally, this schedule gets substituted for Ω_2 in the top-level schedule to yield the valid single appearance schedule $S_p \equiv (2(3A)B(2C))(5D)E$ for Figure 7.2(a).

From S_p and Figure 7.2(a) it is easily verified that $buffer_memory(S_p)$ and

$$\left(\sum_{e \in E} BMLB(e)\right),$$

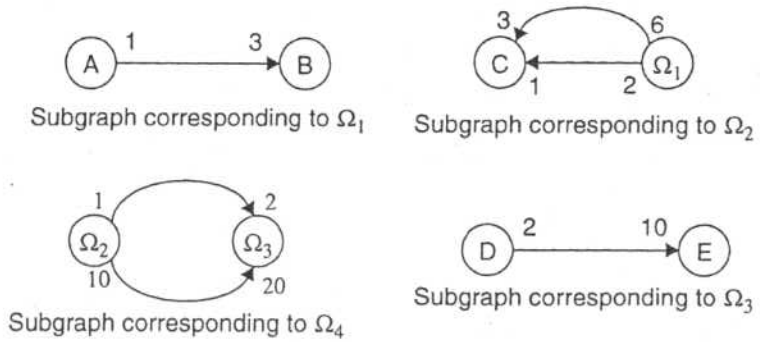


Figure 7.3. A compact representation for the clustering sequence shown in Figure 7.2

where E is the set of edges in Figure 7.2(a), are equal to 43, and thus in the execution of APGAN illustrated in Figure 7.2, a BMLB schedule is constructed.

As seen in the above example, the APGAN approach, as has been defined here, does not uniquely specify the sequence of clusterizations that will be performed, and it does not in general, result in a unique schedule for a given SDF graph. The APGAN technique together with an unambiguous protocol for deciding between adjacent pairs that are tied for the highest repetition count form an **APGAN instance**, which generates a unique schedule for a given graph. For example, one tie-breaking protocol that can be used when actors are labelled alphabetically, as in Figure 7.2, is to choose that adjacent pair that maximizes the sum of the "distances" of the actor labels from the letter "A". If this protocol is used to break the tie between $\{A, B\}$ ("distance sum" is $0 + 1 = 1$) and $\{B, C\}$ (distance sum is $1 + 2 = 3$) in the first clusterization of step of Figure 7.2, then $\{B, C\}$ is chosen.

We say that an adjacent pair is an **APGAN candidate** if it does not introduce a cycle, and its repetition count is greater than or equal to all other adjacent pairs that do not introduce cycles. Thus, an APGAN instance is any algorithm that takes a consistent, acyclic SDF graph as input, repeatedly clusters APGAN candidates, and then outputs the schedule corresponding to a recursive traversal of the resulting cluster hierarchy.

In the following two sections, it is shown that for a consistent, acyclic SDF graph (V, E) that has a BMLB schedule, and that satisfies $del(e) < \eta(e)$ for each $e \in E$, any APGAN instance is guaranteed to obtain a BMLB schedule when applied to this graph. As a consequence, all delay-free graphs, such as that shown in Figure 7.2(a), for which BMLB schedules exist, are handled optimally by any APGAN instance. For example, even if in a certain APGAN instance, $\{B, C\}$ is clustered instead of $\{A, B\}$ in the first clusterization step of Figure 7.2, we are still guaranteed that the final result achieved by that APGAN instance will be a BMLB schedule. To demonstrate the relevance of this optimality result, Section 7.3 will give practical applications to which the result applies. Also, experimental data will be presented in Chapter 8 that suggests that a particular APGAN instance frequently produces excellent results even for applications that do not have BMLB schedules, and it will be shown that it has exhibited encouraging performance on a large collection of complex randomly-generated SDF graphs.

The following fact, which is easily understood from the discussion of the example in Figure 7.2, is fundamental to developing the result on the optimality of APGAN instances.

Fact 7.1: Suppose that G is a connected, consistent, acyclic SDF graph such that $del(e) < \eta(e)$ for each $e \in E$; P is an APGAN instance; and S is the schedule that results when P is applied to G. Then

$$buffer_memory(S) = \sum_{e' \notin E_\Omega} BMLB(e'),$$

where E_Ω is the set of edges that are contained in the subgraphs corresponding to the hierarchical actors $\{\Omega_i\}$ instantiated by P.

For the example of Figure 7.2, E_Ω is the set of six edges that appear in Figure 7.3. It is easily seen that the BMLB values for these edges are, from top to bottom, 3, 6, 2, 10, 2, and 20. Thus, Fact 7.1 states that the schedule obtained from the sequence of clusterizations shown in Figure 7.2 has a buffer memory requirement equal to $3 + 6 + 2 + 10 + 2 + 20 = 43$, which we know is correct from the discussion above.

There are two main parts in the development of the optimality result for APGAN instances. First, a certain class of "proper" clusterizations is defined; it is shown that for delayless graphs, such clusterizations have the property that they do not increase the BMLB values on any edge; and it is shown that under the assumption that a BMLB schedule exists, a clustering operation performed by any APGAN instance is guaranteed to fall in the class of proper clusterizations. Then it is shown that clustering an APGAN candidate cannot transform a graph that has a BMLB schedule into a graph that does not have a BMLB schedule. From these three developments and Facts 6.2 and 7.1, the desired result can be derived easily.

If an efficient data structure, such as a heap, is used to maintain the list of pairwise clustering candidates, then it can be shown that APGAN instances exist with running times that are $O(V^2E)$. The details are beyond the scope of this book.

7.1 Proper Clustering

Definition 7.1: If G is a connected, consistent SDF graph, and $\{X, Y\}$ is an adjacent pair in G that does not introduce a cycle, we say that $\{X, Y\}$ satisfies the **proper clustering condition** in G if for each actor $Z \notin \{X, Y\}$ that is adja-

cent to a member of $\{X, Y\}$, we have that $\rho(\{Z, P\})$ divides $\rho(\{X, Y\})$, for each $P \in \{X, Y\}$ that Z is adjacent to.

In Figure 7.2(a) $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and $\rho(\{B, C\}) = 2$ is divisible by $\rho(\{A, C\}) = 2$, $\rho(\{A, B\}) = 2$, $\rho(\{C, D\}) = 1$, and $\rho(\{B, E\}) = 1$, and thus, $\{B, C\}$ satisfies the proper clustering condition. Conversely, $\rho(\{B, E\})$ is not divisible by $\rho(\{B, C\})$, so $\{B, E\}$ does not satisfy the proper clustering condition.

The motivation for Definition 7.1 is given by Theorem 7.1 below, which establishes that when the proper clustering condition is satisfied, clustering $\{X, Y\}$ does not change the BMLB on any edge, and that when the proper clustering condition is not satisfied, clustering $\{X, Y\}$ increases the BMLB on at least one edge. Thus, a clustering operation that does not satisfy the proper clustering condition cannot be used to derive a BMLB schedule.

To establish Theorem 7.1, we will use the following simple fact about greatest common divisors, which we state here without proof.

Fact 7.2: Suppose that a, b, c are positive integers. If $gcd(\{a, b\})$ divides $gcd(\{a, c\})$, then $gcd(\{a, b, c\}) = gcd(\{a, b\})$; otherwise, $gcd(\{a, b, c\}) < gcd(\{a, b\})$.

Theorem 7.1: Suppose that G is a consistent, connected, delayless SDF graph, and $\{X, Y\}$ is a clusterable adjacent pair in G. If $\{X, Y\}$ satisfies the proper clustering condition, then for each edge e in $G_c \equiv clust(\{X, Y\}, G)$, $BMLB(e') = BMLB(e)$, where e' is the edge in G that corresponds to e. If $\{X, Y\}$ does not satisfy the proper clustering condition, then there exists an edge e in G_c such that $BMLB(e') < BMLB(e)$.

For example, in Figure 6.3, $BMLB((A, B)) = 3$, $BMLB((B, C)) = 3$, and $\mathbf{q}(A, B, C) = (1, 2, 6)$. Figures 7.4(a) and 7.4(b) respectively show $clust(\{A, B\}, G, \Omega)$ and $clust(\{B, C\}, G, \Omega)$, where G denotes the graph of Figure 6.4. In Figure 7.4(a), we see that if $e' = (B, C)$, then $e = (\Omega, C)$, and $BMLB(e) = 6$, while $BMLB(e') = 3$, and thus, $BMLB(e) > BMLB(e')$. In contrast, in Figure 7.4(b), we see that if $e' = (A, B)$, then $e = (A, \Omega)$, and

Figure 7.4. An example used to illustrate Theorem 7.1.

$BMLB(e) = BMLB(e') = 3$. These observations are consistent with Theorem 7.1 since $\rho_G(\{A, B\}) = 1$ divides $\rho_G(\{B, C\}) = 2$, and thus $\{B, C\}$ satisfies the proper clustering condition, while $\rho_G(\{C, B\}) = 2$ does not divide $\rho_G(\{A, B\}) = 1$, and thus $\{A, B\}$ does not satisfy the proper clustering condition.

Proof of Theorem 7.1: First, suppose that $\{X, Y\}$ satisfies the proper clustering condition. Let e be an edge in G_c, and let e' be the corresponding edge in G. If $src(e), snk(e) \neq \Omega$, then $e' = e$, so from Definition 6.1, it follows that $BMLB(e) = BMLB(e')$.

If $src(e) = \Omega$, observe that $snk(e) = snk(e')$ and $src(e') \in \{X, Y\}$, and observe from Fact 4.7 that

$$\rho_{G_c}(\{src(e), snk(e)\}) = gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y), \mathbf{q}_G(snk(e))\}).$$

Thus, since $\{X, Y\}$ satisfies the proper clustering condition, it follows from Fact 7.2 that $\rho_{G_c}(\{src(e), snk(e)\}) = \rho_G(\{src(e'), snk(e')\})$. From Facts 4.8 and 6.3, we conclude that $BMLB(e) = BMLB(e')$. A symmetric argument can be constructed for the case $(snk(e) = \Omega)$. Thus, we have that $BMLB(e) = BMLB(e')$ whenever $\{X, Y\}$ satisfies the proper clustering condition.

If $\{X, Y\}$ does not satisfy the proper clustering condition, then there exists an actor $Z \notin \{X, Y\}$ that is adjacent to some $P \in \{X, Y\}$ such that

$$\rho_G(\{Z, P\}) \text{ does not divide } \rho_G(\{X, Y\}). \tag{7.1}$$

Without loss of generality, suppose that $P = X$ and X is a predecessor of Z (the other possibilities can be handled with symmetric arguments). Let e' be an edge directed from X to Z in G, and let e be the corresponding edge (directed from Ω to Z) in G_c. From Fact 4.7,

$$\rho_{G_c}(\{src(e), snk(e)\}) = gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y), \mathbf{q}_G(snk(e))\}),$$

and thus from (7.1) and Fact 7.2, it follows that

$$\rho_{G_c}(\{src(e), snk(e)\}) < \rho_G(\{src(e'), snk(e')\}).$$

From Facts 4.8 and 6.3, we conclude that $BMLB(e) > BMLB(e')$. ■

The following lemma establishes that if there is an adjacent pair $\{X, Y\}$, X is a predecessor of Y, that introduces a cycle in a delayless SDF graph that has a

BMLB schedule, then there exists an actor $V \notin \{X, Y\}$ that is a predecessor of Y and a descendant of X, such that the repetition count of $\{V, Y\}$ is divisible by the repetition count of $\{X, Y\}$. One interesting consequence of this lemma is that whenever a BMLB schedule exists, the repetition count of an adjacent pair that introduces a cycle cannot exceed the repetition counts of all adjacent pairs that do not introduce cycles. An example is shown in Figure 7.5.

Lemma 7.1: Suppose that G is a connected, delayless, consistent SDF graph that has a BMLB schedule, and e is an edge in G such that $\{src(e), snk(e)\}$ introduces a cycle. Then there exists an actor V in G such that V is a predecessor of $snk(e)$, V is a descendant of $src(e)$; and $\rho_G(\{src(e), snk(e)\})$ divides $\rho_G(\{V, snk(e)\})$.

Proof: Observe that from Theorem 6.1, there exists a BMLB schedule S_R for G that is an R-schedule; since $(\{src(e), snk(e)\})$ introduces a cycle, there is a path $(e_1, e_2, \ldots, e_n)$, $n \geq 2$, from $src(e)$ to $snk(e)$; and from Fact 4.2, $position(src(e), S_R) < position(src(e_n), S_R) < position(snk(e), S_R)$. Thus, there exists a schedule loop $L = (i_0(i_1B_1)(i_2B_2))$ in $(1S_R)$, where B_1 and B_2 are schedule loop bodies such that (a) B_1 contains $src(e)$, and B_2 contains both $src(e_n)$ and $snk(e)$, or (b) B_1 contains both $src(e)$ and $src(e_n)$, and B_2 contains $snk(e)$. Observe that L is simply the innermost schedule loop in $(1S_R)$ that contains $src(e)$, $src(e_n)$, and $snk(e)$.

Without loss of generality, assume that (a) applies — that is, assume that B_1 contains $src(e)$, and B_2 contains both $src(e_n)$ and $snk(e)$. Then there is a schedule loop $L' = (i_0'(i_1'B_1')(i_2'B_2'))$ contained in B_2 such that B_1' contains

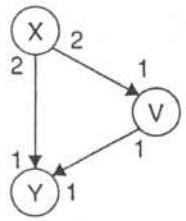


Figure 7.5. An illustration of Lemma 7.1. Here, the repetitions vector is given by $\mathbf{q}(V, X, Y) = (2, 1, 2)$, and $X(2VY)$ is a BMLB schedule. Clearly, $\{X, Y\}$ introduces a cycle. Thus, Lemma 7.1 guarantees that $\rho(\{X, Y\})$ divides $\rho(\{V, Y\})$, and this is easily verified from $\mathbf{q}$.

$src(e_n)$, and B_2' contains $snk(e)$. This is the innermost schedule loop that contains $src(e_n)$ and $snk(e)$, and this schedule loop may be equal to (i_2B_2), or it may be nested in (i_2B_2).

Let I be the product of the iteration counts of all schedule loops in $(1S_R)$ that contain $(i_1B_1)(i_2B_2)$. Similarly, let I' be the product of all schedule loops contained in (i_2B_2) that contain $(i_1'B_1')(i_2'B_2')$. Then, it is easily verified that $max_tokens(e, S_R) = \mathbf{q}_G(src(e))prd(e)/I = TNSE(e)/I$, and $max_tokens(e_n, S_R) = (\mathbf{q}_G(src(e_n))prd(e_n))/(II') = TNSE(e_n)/(II')$.

Since S_R is a BMLB schedule, we have from Fact 6.3 that $\rho_G(\{src(e), snk(e)\}) = I$, and $\rho_G(\{src(e_n), snk(e)\}) = II'$. Thus, $\rho_G(\{src(e), snk(e)\})$ divides $\rho_G(\{src(e_n), snk(e)\})$. Furthermore, since the path $(e_1, e_2, \ldots, e_n)$ originates at $src(e)$, we know that $src(e_n)$ is a descendant of $src(e)$. ■

The following corollary to Lemma 7.1 states that under the hypotheses of Lemma 7.1 (a BMLB schedule exists and $\{src(e), snk(e)\}$ introduces a cycle), we are guaranteed the existence of an adjacent pair $\{V, snk(e)\}$ such that $\{V, snk(e)\}$ does not introduce a cycle, and the repetition count of $\{src(e), snk(e)\}$ divides the repetition count of $\{V, snk(e)\}$.

Corollary 7.1: Assume the hypotheses of Lemma 7.1. Then, there exists a predecessor $V \neq src(e)$ of $snk(e)$ such that $\{V, snk(e)\}$ does not introduce a cycle, and $\rho(\{src(e), snk(e)\})$ divides $\rho(\{V, snk(e)\})$.

Proof: Let $X = src(e)$ and $Y = snk(e)$. From Lemma 7.1, there exists an adjacent pair $\{W_1, Y\}$ such that (a). $\rho(\{X, Y\})$ divides $\rho(\{W_1, Y\})$, and (b). there is a path p_1 from X to W_1. If $\{W_1, Y\}$ introduces a cycle, then again from Lemma 7.1, we have $\{W_2, Y\}$ such that $\rho(\{W_1, Y\})$ divides $\rho(\{W_2, Y\})$, and there is a path p_2 from W_1 to W_2. Furthermore, $W_2 \neq X$, since $(W_2 = X)$ implies that $\langle(p_1, p_2)\rangle$ is a cycle, and thus that G is not acyclic.

If $(\{W_2, Y\})$ introduces a cycle, then from Lemma 7.1, we have $(\{W_3, Y\})$ such that $\rho(\{W_2, Y\})$ divides $\rho(\{W_3, Y\})$, and there is a path p_3 from W_2 to W_3. Furthermore $W_3 \neq X$, since otherwise $\langle(p_1, p_2, p_3)\rangle$ is a cycle in G; similarly, $W_3 \neq W_1$, since otherwise $\langle(p_2, p_3)\rangle$ is a cycle. Continuing this process, we obtain a sequence of *distinct* actors $(W_1, W_2, \ldots)$. Since the W_is are distinct and we are assuming a finite graph, we cannot continue generating W_is indefinitely. Thus, eventually, we will arrive at a W_n such that $(\{W_n, Y\})$ does not introduce a cycle. Furthermore, by construction, $\rho(\{X, Y\})$ divides

$\rho(\{W_1, Y\})$, and for $i \in \{1, 2, \ldots, (n-1)\}$, $\rho(\{W_i, Y\})$ divides $\rho(\{W_{i+1}, Y\})$. It follows that $\rho(\{X, Y\})$ divides $\rho(\{W_n, Y\})$. ■

As a consequence of Corollary 7.1, we can be sure that given an APGAN candidate $\{X, Y\}$ in an SDF graph that has a BMLB schedule, no other adjacent pair has a higher repetition count. As an example consider Figure 7.6(a), and suppose that the SDF parameters on the graph edges are such that $(\{A, B\})$ is an APGAN candidate — that is, $(\{A, B\})$ does not introduce a cycle and maximizes $\rho(*)$ over all adjacent pairs that do not introduce cycles. Since $(\{B, C\})$ introduces a cycle, the assumption that $(\{A, B\})$ is an APGAN candidate is not sufficient to guarantee that $\rho(\{B, C\}) \le \rho(\{A, B\})$. However, Theorem 7.2 below guarantees that under the additional assumption that Figure 7.6(a) has a BMLB schedule, $\rho(\{B, C\})$ is guaranteed not to exceed $\rho(\{A, B\})$.

Figure 7.6(b) shows a case where this additional assumption is violated. Here, $\mathbf{q}(A, B, C, D) = (2, 4, 8, 1)$. It is easily seen that four invocations of B must fire before a single invocation of C can fire, and thus for any valid schedule S, $max_tokens((B, C), S) \ge 4 \times 2 = 8 > BMLB((B, C))$; consequently, Figure 7.6(b) cannot have a BMLB schedule. It is also easily verified that among the three adjacent pairs in Figure 7.6(b) that do not introduce cycles, $\{A, B\}$ is the only APGAN candidate, and $\rho(\{B, C\}) = 4$, while $\rho(\{A, B\}) = 2$. Thus, the conclusion of Theorem 7.2 does not generally hold if we relax the assumption that the graph in question has a BMLB schedule.

Theorem 7.2: Suppose that G is a connected, delayless SDF graph that has a BMLB schedule, and p is an APGAN candidate in G. Then for all adjacent pairs p' in G, $\rho(p) \ge \rho(p')$.

Proof: (By contraposition.) Suppose that $\rho(p') > \rho(p)$. Then since p is an APGAN candidate, p' must introduce a cycle. From Corollary 7.1, there exists an

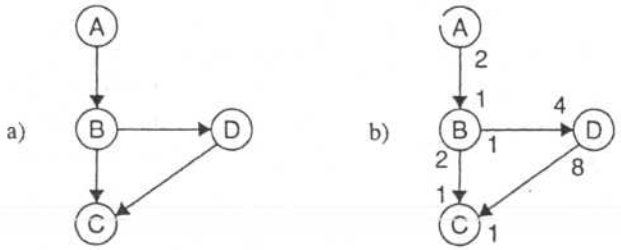


Figure 7.6. Examples used to illustrate Theorem 7.2.

adjacent pair p'' such that p'' does not introduce a cycle, and $\rho(p')$ divides $\rho(p'')$. It follows that $\rho(p'') > \rho(p)$. Since p'' does not introduce a cycle, p cannot be an APGAN candidate. ■

Lemma 7.2: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, $R \subseteq V$ is a subset of actors such that $C \equiv subgraph(R)$ is connected, and $X, Y, Z \in R$. Then

$$(gcd(\{\mathbf{q}_C(X), \mathbf{q}_C(Y)\}) \text{ divides } gcd(\{\mathbf{q}_C(Y), \mathbf{q}_C(Z)\})) \Rightarrow$$
$$(gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y)\}) \text{ divides } gcd(\{\mathbf{q}_G(Y), \mathbf{q}_G(Z)\})).$$

Proof: Suppose that $(gcd(\{\mathbf{q}_C(Y), \mathbf{q}_C(Z)\}))/(gcd(\{\mathbf{q}_C(X), \mathbf{q}_C(Y)\})) = k$, for some positive integer k. Then, from Fact 4.5,

$$\begin{aligned}
&(gcd(\{\mathbf{q}_G(Y), \mathbf{q}_G(Z)\}))/(gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y)\})) \\
&= (gcd(\{q_G(R)\mathbf{q}_C(Y), q_G(R)\mathbf{q}_C(Z)\}))/(gcd(\{q_G(R)\mathbf{q}_C(X), q_G(R)\mathbf{q}_C(Y)\})) \\
&= (gcd(\{\mathbf{q}_C(Y), \mathbf{q}_C(Z)\}))/(gcd(\{\mathbf{q}_C(X), \mathbf{q}_C(Y)\})) \\
&= k. ■
\end{aligned}$$

The following lemma states that in a connected SDF graph that contains exactly three actors, and that has a BMLB schedule, the repetition count can exceed unity for at most one adjacent pair. For example, consider the three-actor graph in Figure 7.7. Here, the repetitions vector is given by $\mathbf{q}(A, B, C) = (6, 2, 3)$, and $(2(3A)B)(3C)$ is a BMLB schedule. The two pairs of adjacent actors $\{A, B\}$ and $\{B, C\}$ have repetition counts of 2 and 1, respectively. Thus, we see that only one adjacent pair has a repetition count that exceeds unity.

Lemma 7.3: Suppose that (a). G is a connected, consistent, delayless SDF graph that consists of exactly three distinct actors X, Y and Z; (b). X is a predecessor of Y; (c). $Z \notin \{X, Y\}$ is adjacent to $P \in \{X, Y\}$; (d). $\rho_G(\{X, Y\}) \ge \rho_G(\{P, Z\})$; and (e). G has a BMLB schedule. Then, $\rho_G(\{P, Z\}) = 1$.

Proof: For simplicity, assume that $P = Y$, and that Z is a successor of Y. The other three possible cases — ($P = Y$, Z is a predecessor of Y), and ($P = X$, Z is a predecessor or successor of X) — can be handled by simple adaptations of this argument.

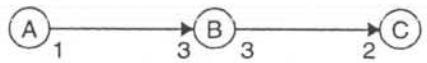


Figure 7.7. An illustration of Lemma 7.3.

Let e_{xy} be an edge directed from X to Y, and let e_{yz} be an edge directed from Y to Z. From Theorem 6.1, there exists a BMLB R-schedule S_R for G. Since G contains only three actors, G has exactly two R-schedules, and it is easily verified that either S_R is of the form $(i_1X)(i_2(i_3Y)(i_4Z))$, or it has the form $(j_1(j_2X)(j_3Y))(j_4Z)$.

If $S_R = (i_1X)(i_2(i_3Y)(i_4Z))$, then $max_tokens(e_{xy}, S_R) = TNSE(e_{xy})$, and thus from Fact 6.3, we have that $TNSE(e_{xy}) = TNSE(e_{xy})/\rho(\{X, Y\})$, which implies that $\rho(\{X, Y\}) = 1$. It follows that $\rho(\{Y, Z\}) = 1$ from assumption(d).

Conversely, suppose that $S_R = (j_1(j_2X)(j_3Y))(j_4Z)$. Then $max_tokens(e_{yz}, S_R) = TNSE(e_{yz})$, so from Fact 6.3, we have that
$TNSE(e_{yz}) = TNSE(e_{yz})/\rho(\{Y, Z\})$,
which implies the desired result. ■

The following theorem guarantees that whenever an APGAN instance performs a clustering operation on a top-level graph that has a BMLB schedule, the adjacent pair selected satisfies the proper clustering condition in the top-level graph. For example in Figure 7.2(a), $\{A, B\}$ and $\{B, C\}$ are APGAN candidates, and it is easily verified from the repetitions vector $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$ that both of these adjacent pairs satisfy the proper clustering condition in Figure 7.2(a). Similarly, for Figure 7.2(b) we have $\mathbf{q}(\Omega_1, C, D, E) = (2, 4, 5, 1)$, and thus $\{\Omega_1, C\}$ is the only APGAN candidate. Thus, Theorem 7.3 guarantees that $\{\Omega_1, C\}$ satisfies the proper clustering condition in Figure 7.2(b).

Theorem 7.3: Suppose that G is a connected, consistent, delayless SDF graph; a BMLB schedule exists for G; and $\{X, Y\}$ is an APGAN candidate in G. Then $\{X, Y\}$ satisfies the proper clustering condition in G.

Proof: Let $Z \notin \{X, Y\}$ be an actor that is adjacent to some $P \in \{X, Y\}$; let $C = subgraph(\{X, Y, Z\})$, and observe from Fact 4.4 that C has a BMLB schedule. From Theorem 7.2, $\rho_G(\{Z, P\}) \le \rho_G(\{X, Y\})$, and from Fact 4.5, it follows that $\rho_C(\{Z, P\}) \le \rho_C(\{X, Y\})$. Applying Lemma 7.3 to the three-actor graph C, we see that $\rho_C(\{Z, P\}) = 1$, and thus from Lemma 7.2, $\rho_G(\{Z, P\})$ divides $\rho_G(\{X, Y\})$. ■

7.2 The Optimality of APGAN for a Class of Graphs

In this section, we use the main results of Section 7.1 to show that for any acyclic SDF graph (V, E) that has a BMLB schedule, and that satisfies

$del(e) < \eta(e)$, for all $e \in E$, any APGAN instance is guaranteed to construct a BMLB schedule.

In Section 7.1, it was shown that clustering an adjacent pair that satisfies the proper clustering condition does not change the BMLB on an edge. However, to derive a BMLB schedule whenever one exists, it is not sufficient to simply ensure that each clusterization step selects an adjacent pair that satisfies the proper clustering condition. This is because although clustering an adjacent pair that satisfies the proper clustering condition preserves the BMLB value on each edge, it does not necessarily preserve the existence of a BMLB schedule.

Consider the SDF graph in Figure 7.8(a) ($\mathbf{q}(A, B, C, D, E, F) = (3, 5, 10, 10, 5, 2)$). It is easily verified that $(3A)(5B(2DC)E)(2F)$ is a BMLB schedule. Also, observe that $\rho(\{A, F\}) = \rho(\{A, B\}) = \rho(\{E, F\}) = 1$, and thus, $\{A, F\}$ satisfies the proper clustering condition. Figure 7.8(b) shows $clust(\{A, F\}, G, \Omega)$, where G denotes the graph of Figure 7.8(a). In Figure 7.8(b), we see that due to the path $((D, E), (E, \Omega), (\Omega, B), (B, C))$, D must fire 10 times before a single invocation of C can fire, and thus $max_tokens((D, C), S) \ge 10 > BMLB((D, C))$, if S is a valid schedule for Figure 7.8(b). Thus, Figure 7.8(b) cannot have a BMLB schedule, and we see that even though $\{A, F\}$ satisfies the proper clustering condition in Figure 7.8(a), clustering this adjacent pair does not preserve the existence of a BMLB schedule.

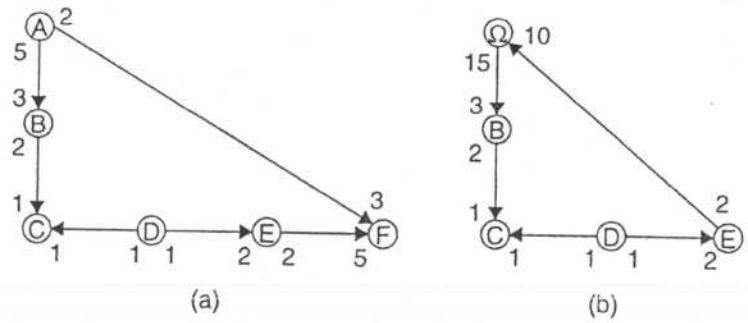


Figure 7.8. An example of how clustering an adjacent pair that satisfies the proper clustering condition can cancel the existence of a BMLB schedule.

Fortunately, the assumption that the adjacent pair being clustered has maximum repetition count is sufficient to preserve the existence of a BMLB schedule. Thus, clustering an APGAN candidate always preserves the existence of a BMLB schedule.

Theorem 7.4: Suppose that $G = (V, E)$ is a connected, consistent, delayless SDF graph with $|V| > 1$; G has a BMLB schedule; and $\{X, Y\}$ is an APGAN candidate in G. Then $clust(\{X, Y\}, G)$ has a BMLB schedule.

Proof: We assume without loss of generality that X is a predecessor of Y, and we prove this theorem by induction on $|V|$. Clearly, the theorem holds trivially for $|V| = 2$, since in this case, $clust(\{X, Y\}, G)$ contains no edges. Now suppose that the theorem holds for $|V| = 2, 3, \ldots, k$, and consider the case $|V| = (k+1)$.

Define $G_c = clust(\{X, Y\}, G, \Omega)$, and let S_R be a BMLB R-schedule for G; the existence of such a schedule is guaranteed by Theorem 6.1. Since S_R is an R-schedule and $|V| > 2$, S_R is of the form $(i_1 B_1)(i_2 B_2)$.

Now suppose that $X, Y \in actors(B_1)$, and let $C_1, C_2, \ldots, C_n$ denote the connected components of $subgraph(actors(B_1))$. Observe that from Fact 4.4, $S_i = projection((i_1 B_1), C_i)$ is a BMLB schedule for each C_i. Let C_j denote that connected component that contains X and Y. Then, since $|C_j| \le k$, we can apply Theorem 7.4 with $|V| = |C_j|$ to obtain a BMLB schedule S^* for $clust(\{X, Y\}, subgraph(C_j))$, and from Theorem 4.5, we can assume without loss of generality that $J(S^*) = J(S_j)$. Then, it is easily verified that $S_1 S_2 \ldots S_{j-1} S^* S_{j+1} S_{j+2} \ldots S_n (i_2 B_2)$ is a BMLB schedule for G_c. A similar argument can be applied to establish the existence of a BMLB schedule for G_c when $X, Y \in actors(B_2)$.

Now suppose that $X \in actors(B_1)$ and $Y \in actors(B_2)$, and let e_{xy} be an edge directed from X to Y. Also, let E_c denote the set of edges in G_c, and for each $e \in E_c$, let e' denote the corresponding edge in G. Clearly $max_tokens(e_{xy}, S_R) = TNSE(e_{xy})$, and thus, since S_R is a BMLB schedule, we have from Fact 6.3 that $\rho_G(\{X, Y\}) = 1$. From Theorem 7.2, it follows that $\rho_G\{X', Y'\} = 1$ for all adjacent pairs $\{X', Y'\}$ in G. Thus, from Fact 6.3,

$$BMLB(e) = TNSE(e, G) \text{ for all } e \in E. \tag{7.2}$$

Let $(X_1, X_2, \ldots, X_n)$ be any topological sort for G_c. Then clearly, $S_c = (\mathbf{q}_{G_c}(X_1))(\mathbf{q}_{G_c}(X_2))\ldots(\mathbf{q}_{G_c}(X_n))$ is a valid single appearance schedule for G_c, and

$$\begin{aligned}
buffer_memory(S_c) &= \sum_{e \in E_c} TNSE(e, G_c) \\
&= \sum_{e \in E_c} TNSE(e', G) && \text{(from Fact 4.8)} \\
&= \sum_{e \in E_c} BMLB(e') && \text{(from (7.2))} \\
&= \sum_{e \in E_c} BMLB(e). && \text{(from Theorems 7.1 and 7.2)}
\end{aligned}$$

Thus, S_c is a BMLB schedule for G_c. ■

We are now able to establish the result on the optimality of APGAN.

Lemma 7.4: Suppose that $G = (V, E)$ is a connected, consistent, delayless SDF graph that has a BMLB schedule; P is an APGAN instance; and $S_P(G)$ is the schedule obtained by applying P to G. Then $S_P(G)$ is a BMLB schedule for G.

Proof: By definition, P repeatedly clusters APGAN candidates until the top-level graph consists of only one actor. From Theorem 7.3, the first adjacent pair p_1 clustered when P is applied to G satisfies the proper clustering condition, and thus from Theorem 7.4, the top level graph T_1 that results from the first clustering operation has a BMLB schedule. Since T_1 has a BMLB schedule we can again apply Theorems 7.3 and 7.4 to conclude that the second adjacent pair p_2 clustered by P satisfies the proper clustering condition, and that the top-level graph T_2 obtained from clustering p_2 in T_1 has a BMLB schedule. Continuing in this manner successively for $p_3, p_4, \ldots, p_n$, where n is the total number of adjacent pairs clustered when P is applied to G, we conclude that each adjacent pair clustered by P satisfies the proper clustering condition. Thus, from Theorem 7.1, $BMLB(e') = BMLB(e)$, whenever e' and e are corresponding edges associated with a clusterization step of P. It follows from Fact 7.1 that

$$buffer_memory(S_P(G)) = \sum_{e \in E} BMLB(e),$$

and thus $S_P(G)$ is a BMLB schedule for G. ■

The following theorem gives the general specification of the optimality of APGAN instances.

Theorem 7.5: Suppose that $G = (V, E)$ is a connected, consistent, acyclic SDF graph that has a BMLB schedule; $del(e) < \eta(e)$ for all $e \in E$; P is an APGAN instance; and $S_P(G)$ is the schedule obtained by applying P to G. Then $S_P(G)$ is a BMLB schedule for G.

Proof: Let G' denote the delayless version of G, and let P' be the APGAN instance that returns $S_P(G)$ if the input graph is G', and returns $S_P(G_I)$ otherwise, where G_I is the input graph. Clearly P' is an APGAN instance since edge delays do not affect the repetition counts of adjacent pairs. From Lemma 7.4 and Fact 6.2, $S_{P'}(G')$ is BMLB schedule for G. But by construction, $S_{P'}(G') = S_P(G)$. ■

Figure 7.9 shows what can "go wrong" in trying to achieve the BMLB with APGAN when the assumption that $del(e) < \eta(e)$ is not satisfied for all edges. In the SDF graph of (a), $\mathbf{q}(A, B, C, D) = (1, 1, 1, 1)$, and thus all adjacent pairs have the same repetition count. Thus, two possible clusterization sequences by an APGAN instance for this graph are $\{W, Y\}$ followed by $\{X, Z\}$ (shown in (b)), and $\{W, X\}$ followed by $\{Y, Z\}$ (shown in (c)). From (b) and (c), we see that the schedules resulting from these two clusterization sequences are (ignoring all one-iteration loops), respectively, $YWZX$ and $YZXW$. Here, the former schedule has a buffer memory requirement of 5, while the latter schedule has a buffer memory requirement of 4 since the sink actor fires before the source actor for each edge that has unit delay. Thus, we see that different APGAN instances will

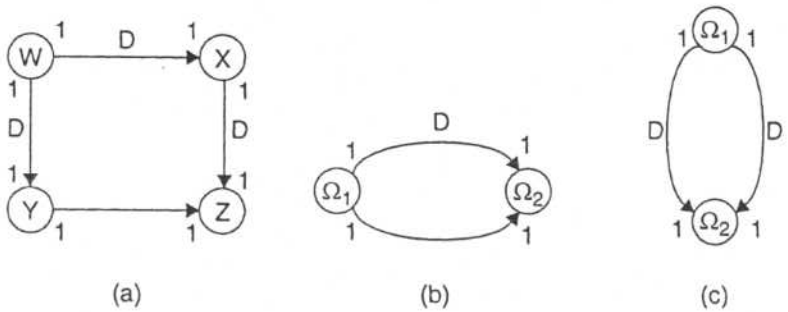


Figure 7.9. An example of how an APGAN instance may fail to achieve the BMLB when $del(e) < \eta(e)$ does not hold for every edge e.

in general produce different buffer memory requirements when applied to Figure 7.9(a).

7.3 Examples

7.3.1. Tree-structured Filter Bank

Figure 7.10 shows an SDF graph abstraction of a uniform-tree structured QMF filterbank [Vaid93]. This type of filterbank is commonly used in practice for audio coding applications. Filterbanks like this, even with arbitrary "depth", where depth is defined to be the logarithm of the number of channels in the middle of the graph, fall into the class of SDF graphs that have BMLB schedules; hence, APGAN will always return an optimal buffer schedule for these graphs.

7.3.2. Satellite Receiver

In the example above, the graph had a very symmetric and regular topology. Figure 7.11 shows an example where the topology is not as regular, but is still in the class of SDF graphs that have a BMLB schedule. The graph is an abstraction for a satellite receiver implementation and is taken from [Ritz95]. The graph is annotated with the produced/consumed numbers wherever they are different from unity. It is interesting to note that a shared-buffer implementation of the flat single appearance schedule for this graph would require a buffer of size 2040 [Ritz95] while APGAN generates a BMLB schedule having a total buffering requirement of 1540 (using a buffer on every edge of-course).

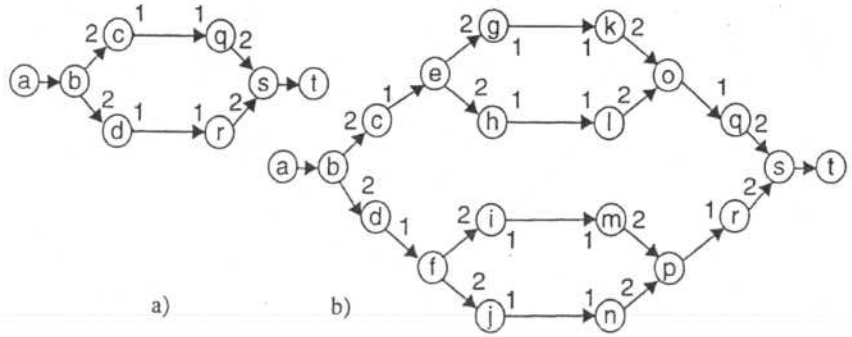


Figure 7.10. SDF graph for a uniform-tree filterbank. a) Depth 1 filterbank, b) Depth 2 filterbank. The produced/consumed numbers not specified are all unity.

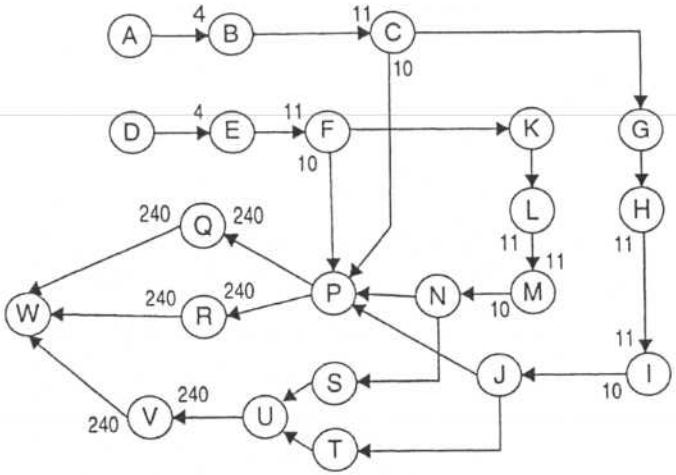


Figure 7.11. SDF abstraction for satellite receiver application from [Ritz95]

7.3.3. Chain Structured Graph with Irregular Rate Changes

When sample rates are irregular, APGAN can do poorly even if the topology is very simple. For the graph shown in figure 7.12, it can be verified that APGAN will construct the schedule $(9A)(4(3BC)2D)$, which has a buffering cost of 43, while the optimal schedule, returned by GDPPO is $(3(3A)(4B))(4(3C)(2D))$, which has a cost of 30. The optimal schedule would also be returned by RPMC (without GDPPO) in this case since edge BC is where the minimum amount of data is transferred in a complete period of the schedule.

The tables in Chapter 8 have more examples illustrating the performance of the APGAN algorithm. The tables have both the performance on practical examples and performance on random graphs. As noted there, APGAN generally performs well when the topology or rate changes are fairly regular. Such regularity arises frequently in practical multirate SDF graphs. In graphs that contain significant irregularity, such as random graphs and the non-uniform filterbanks, RPMC usually performs much better than APGAN. Hence, these two heuristics complement each other well.

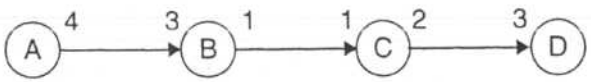


Figure 7.12. A chain-structured SDF graph.

8

Experiments

Table 8.1 shows the results of applying GDPPO to the schedules generated by APGAN and RPMC on several practical SDF systems. The columns labeled "% Impr." show the percentage of buffer memory reduction obtained by GDPPO. The QMF tree filter banks fall into a class of graphs for which APGAN is guaranteed to produce optimal results (see Chapter 7), and thus there is no room for GDPPO to produce improvement when APGAN is applied to these two examples. Overall, GDPPO produces an improvement in 11 out of the 14 heuristic/application combinations. A "significant" (greater than 5%) improvement is obtained in 9 of the 14 combinations; the mean improvement over all 14 combinations is 9.9%; and from the CD-DAT and DAT-CD examples, we see that it is possible to obtain very large reductions in the buffer memory requirement with GDPPO.

Table 8.2 shows experimental results on the performance of APGAN and RPMC for several practical examples of acyclic, multirate SDF graphs. The column titled "average random" represents the average buffer memory requirement obtained by considering 100 **random schedules**. A random schedule is generated by generating a random topological sort. Corresponding to this topological sort is a flat single appearance schedule; this is a BMUB schedule. Then GDPPO is applied to this schedule to get a single appearance schedule whose buffer memory requirement is less than or equal to that of all single appearance schedules having this particular topological ordering of the actors. All of the systems shown in the table are acyclic graphs. The data for APGAN and RPMC also includes the effect

Table 8.1 Performance of GDPPO on several practical SDF systems.

Application	Apgan only	Apgan + Gdppo	% Impr.	Rpmc only	Rpmc + Gdppo	% Impr.
Nonuniform filter bank (1/3, 2/3 splits, 4 channels)	153	137	10.5	131	128	2.34
Nonuniform filter bank (1/3, 2/3 splits, 6 channels)	856	756	11.7	690	589	14.6
QMF tree filter bank (8 channels)	78	78	0	92	87	5.43
QMF tree filter bank (16 channels)	166	166	0	218	200	8.26
Two-stage fractional decimation system	140	119	15.0	133	133	0
CD-DAT sample rate conversion	396	382	3.54	535	400	25.2
DAT-CD sample rate conversion	205	182	11.2	275	191	30.5

of GDPPO. As can be seen, APGAN achieves the BMLB on 5 of the 9 examples, outperforming RPMC in these cases. Particularly interesting are the last three examples in the table, which illustrate the performance of the two heuristics as the graph sizes are increased. The graphs represent a symmetric tree-structured QMF filterbank with differing depths. APGAN constructs a BMLB schedule for each of these systems while RPMC generates schedules that have buffer memory requirements about 1.2 times the optimal. Conversely, the third and fourth entries show that RPMC can outperform APGAN significantly on graphs that have more irregular rate changes. These graphs represent nonuniform filterbanks with differing depths. In the table, for each example, the cell corresponding to the heuristic that gave the best buffer memory requirement has been shaded. If the best performer equals the BMLB, then the shading is the same as the BMLB column; otherwise a darker shade has been used.

Table 8.3 shows more detailed statistics for the performance of randomly obtained topological sorts. For example, the column titled "APGAN < random"

Table 8.2 Performance of the two heuristics on various acyclic graphs.

System	BMUB	BMLB	Apgan	Rpmc	Avg. Rand.	# vert./ # edges
Fractional decimation	61	47	47	52	52	26/30
Laplacian pyramid	115	95	99	99	102	12/13
Nonuniform filterbank (1/3,2/3 splits) (4 channels)	466	85	137	128	172	27/29
Nonuniform filterbank (1/3,2/3 splits) (6 channels)	4853	224	756	589	1025	43/47
QMF nonuniform-tree filterbank	284	154	160	171	177	42/45
QMF filterbank (one-sided tree)	162	102	108	110	112	20/22
QMF analysis only	248	35	35	35	43	26/25
QMF Tree filterbank (4 channels)	84	46	46	55	53	32/34
QMF Tree filterbank (8 channels)	152	78	78	87	93	44/50
QMF Tree filterbank (16 channels)	400	166	166	200	227	92/106

represents the number of random schedules (again, these are obtained by starting with a BMUB schedule for a random topological sort and applying GDPPO to this schedule) that had a buffer memory requirement greater than that obtained by APGAN. The last two columns give the mean number of random schedules needed to outperform these heuristics. A dash indicates that no random schedules were found that had a buffer memory requirement lower that obtained by the corresponding heuristic.

While the above results on practical examples are encouraging, these heuristics have also been tested on a large number of randomly generated 50-actor

Table 8.3 Performance of 100 random schedules against the heuristics

Comparison with random schedules (100 trials)	Apgan < random	Apgan = random	Rpmc < random	Rpmc = random	Avg. to beat Apgan	Avg. to beat Rpmc
Fractional decimation	92%	8%	54%	13%	----	3
Laplacian pyramid	74%	26%	74%	26%	----	----
Nonuniform filterbank (1/3,2/3 splits) (4 channels)	100%	0%	100%	0%	----	----
Nonuniform filterbank (1/3,2/3 splits) (6 channels)	100%	0%	100%	0%	----	----
QMF nonuniform-tree filterbank	100%	0%	81%	7%	----	8
QMF filterbank (one-sided tree)	100%	0%	77%	23%	----	----
QMF analysis only	99%	1%	99%	1%	----	----
QMF Tree filterbank (4 channels)	100%	0%	16%	13%	----	1.4
QMF Tree filterbank (8 channels)	100%	0%	87%	3%	----	9.1
QMF Tree filterbank (16 channels)	100%	0%	96%	1%	----	22.3

acyclic SDF graphs. These graphs were sparse, having about 100 edges on average. The SDF parameters were chosen randomly according to the following rules. Firstly, it is determined whether a parameter is a "free variable" or not; it is a free variable if assigning an arbitrary number to the parameter does not lead to sample rate inconsistency. If the parameter is a "free variable", then with probability 0.5, it is set to 1, and with probability 0.5, it is set to a uniformly generated random number between 1 and 10. Table 8.4 summarizes the performance of these heuristics, both against each other, and against randomly generated schedules. As can be seen, RPMC outperforms APGAN on these random graphs almost two-thirds

Table 8.4 Performance of the two heuristics on random graphs

RPMC < APGAN	63%
APGAN < RPMC	37%
RPMC < min(2 random)	83%
APGAN < min(2 random)	68%
RPMC < min(4 random)	75%
APGAN < min(4 random)	61%
min(RPMC,APGAN) < min(4 random)	87%
RPMC < APGAN by more than 10%	45%
RPMC < APGAN by more than 20%	35%
APGAN < RPMC by more than 10%	23%
APGAN < RPMC by more than 20%	14%

of the time.These heuristics were compared against 2 random schedules because measurements of the actual running time on 50-vertex graphs showed that we can construct and examine approximately 2 random schedules and post-optimize it by GDPPO in the time it takes for either APGAN or RPMC to construct its schedule and have it post-optimized by GDPPO. The comparison against 4 random schedules shows that in general, the relative performance of these heuristics goes down if a large number of random schedules are inspected. Of course, this also entails a proportionate increase in running time. However, as shown on practical examples already, it is unlikely that even picking a large number of schedules randomly will give better results than these heuristics since practical graphs usually have a significant amount of structure (as opposed to random graphs) that the heuristics can exploit well. Thus, the comparisons against random graphs give a worst case estimate of the performance that can be expected from these heuristics.

The experimental results presented in this chapter show that APGAN and RPMC complement each other. For the practical SDF graphs that were examined, APGAN performs well on graphs that have a simple structure topologically and regular rate changes, like the uniform QMF filterbanks, and RPMC performs well on graphs that have more irregular rate changes and irregular topologies. Since

large random graphs can be expected to consistently have irregular rate changes and topologies, the average performance on random graphs of RPMC is better than APGAN by a wide margin — although, from the last two rows of Table 8.4, it can be seen that there is a significant proportion of random graphs for which APGAN outperforms RPMC by a margin of over 10%, which suggests that APGAN is a useful complement to RPMC even when mostly irregular graphs are encountered. However, the main advantage of adopting both APGAN and RPMC as a combined solution arises from complementing the strong performance of RPMC on general graphs with the formal properties of APGAN, as specified by Theorem 7.5, and the ability of APGAN to exploit regularity that arises frequently in practical applications.

There is a variation of APGAN that achieves significantly better performance on random graphs than the original version, although still significantly worse performance as compared to RPMC. This variation arises from changing the "priority function" associated with an edge e from $\rho(\{src(e), snk(e)\})$ to the product

$$TNSE(e) \times \rho(\{src(e), snk(e)\}). \tag{8.1}$$

In other words, the variation repeatedly clusters the source and sink vertices of edges that maximize the measure given by (8.1). Thus, an adjacent pair is given more weight if a large amount of data is transferred between it as compared to other adjacent pairs.

We have found that on the random graphs that were used to generate Table 8.4, this modification of APGAN outperforms two random schedules ("min(2 random)") 76.5 percent of the time, which indicates a level of performance intermediate to APGAN and RPMC. Furthermore, its performance equaled the performance of APGAN on all of the practical examples except the six channel nonuniform filter bank, where it achieved a buffer memory requirement of 696 (8% better than APGAN), and the four channel nonuniform filter bank, where it achieved 136 (0.7% better than APGAN).

Interestingly, however, the modification of APGAN corresponding to (8.1) does not preserve the formal properties specified by Theorem 7.5. This is easily seen from the example in Figure 8.1. Here, $\mathbf{q}(W, X, Y) = (2, 2, 1)$, and thus $\rho(\{W, X\}) = 2$ and $\rho(\{W, Y\}) = 1$, and if we let $\hat{\rho}$ denote the measure defined by (8.1), then $\hat{\rho}(\{W, X\}) = 2 \times 2 = 4$, while $\hat{\rho}(\{W, Y\}) = 6 \times 1 = 6$. We see then that APGAN clusters $\{W, X\}$ in its first clusterization step, which

leads to the final schedule $(2WX)Y$, and a buffer memory requirement of 7, while in the variation of APGAN, $\{W, Y\}$ is clustered first, and the resulting schedule $(2W)Y(2X)$ gives a buffer memory requirement 8. It is easily verified the BMLB for this graph is 7, and thus, APGAN generates a BMLB schedule, while the variation generates a suboptimal result.

Thus, the variation of APGAN introduces a trade-off between provable optimality for a class of graphs, and average-case performance. Since we are proposing to complement a heuristic — RPMC — whose average case performance significantly outweighs that of both APGAN and its variation, it is intuitively more appealing to choose the original version of APGAN since it adds a feature that RPMC lacks — optimality for a restricted, but useful, class of graphs. For the practical examples that were examined, the variation of APGAN outperformed the original APGAN only in cases where RPMC outperformed both APGAN and the APGAN variation, and thus adopting the new version of APGAN does not improve the final result of any of these examples when a combined solution with RPMC is employed.

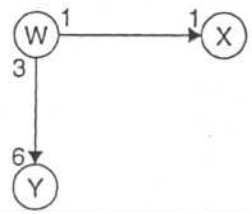


Figure 8.1. An example in which APGAN achieves the BMLB, but the modified version corresponding to (8.1) does not.

9

Open Issues

This book has presented the state of the art in SDF scheduling theory. A formal theory for constructing and manipulating looped schedules from SDF representations of digital signal processing algorithms has been given, and based on this theory, techniques have been presented for compiling SDF programs into efficient code for programmable processors. The techniques have focused on the minimization of code size, and the minimization of the buffer memory requirement. We have defined a class of code-size-minimizing schedules called *single appearance schedules*. There are two central themes. The first is a uniprocessor scheduling framework that constructs single appearance schedules whenever they exist, and when single appearance schedules do not exist, guarantees optimal code size for all actors that are not contained in a certain type of subgraph called a *tightly interdependent subgraph*. The second theme consists of techniques that construct single appearance schedules for acyclic graphs that minimize data memory size, and an extensive experimental evaluation of these techniques on practical and random graphs. These techniques can easily be incorporated into the scheduling framework described in Chapter 5. Despite the focus on single appearance scheduling, many results of a more general nature, such as clustering theory, and the problem of minimum-buffer scheduling (not necessarily single appearance schedules) have also been presented.

In this remainder of this chapter, we discuss a number of problems that remain open in the area of compiling SDF graphs.

9.1 Tightly Interdependent Graphs

Loose interdependence algorithms guarantee optimal code size for each actor that does not lie in a tightly interdependent subgraph, and they guarantee that the number of appearances of each actor within a tightly interdependent subgraph is determined entirely by the tight scheduling algorithm. This book does not discuss any techniques for scheduling tightly interdependent components that give guarantees on the compactness of the schedule. If developed, such techniques can be incorporated into the tight scheduling algorithm component of the loose interdependence scheduling framework without affecting the performance of the other three component algorithms.

One direction of study in the problem of compactly scheduling tightly interdependent graphs is the application of *retiming*. Retiming was proposed by Leiserson *et al.* [Leis83] as a technique for minimizing the clock period of synchronous digital circuits. Extension of the retiming concept to general SDF graphs was discussed by Lee in [Lee86]; and Zivojnovic *et al.* analyze properties of retimed SDF graphs and they formulate an integer linear programming solution to the problem of minimizing the total delay count of an SDF graph through retiming [Zivo93].

In an SDF graph, retiming can be viewed as rearranging the delays in accordance with certain constraints. As a simple example of retiming, and how it can improve the scheduling of tightly interdependent subgraphs, consider the example of Figure 9.1. Figure 9.1(a) shows a tightly interdependent SDF graph, and Figure 9.1(b) shows how moving the delay on the edge $A \rightarrow B$ to the edge $B \rightarrow A$ results in a graph that has a single appearance schedule. This application of retiming can be implemented by firing B once as a *preamble* to the periodic schedule. However, the code to construct this preamble will negate the advantage

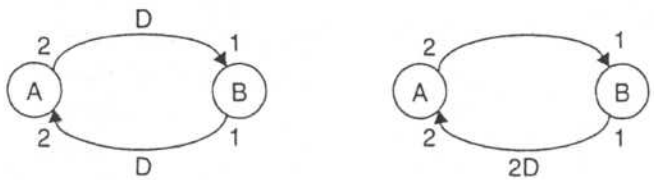


Figure 9.1. An example of how retiming can lead to more compact schedules of SDF graphs.

of having a more compact periodic schedule. Alternatively, since the transformation in computation may lead simply to a transient that diminishes with time, it may be valid to ignore the preamble and directly implement a periodic schedule for the retimed graph. In such cases, applications of retiming such as the example of Figure 9.1 can improve code size compactness for tightly interdependent graphs. Thus, it would be useful if we could efficiently determine when a tightly interdependent SDF graph can be retimed into an SDF graph that has a single appearance schedule, and if we could determine appropriate retimings for such cases.

9.2 Buffering

It would be desirable to extend the techniques for determining minimum buffer single appearance schedules for acyclic graphs to cyclic graphs.

Systematic assessment of scheduling trade-offs between the code size and the buffer memory requirement is another area for further study. For example, if a single appearance schedule has been constructed, and the resulting code does not fully occupy the available program memory, then we would like to know how the remaining program memory can be utilized to expand the schedule in such a way that the buffer memory requirement is minimized. Alternatively, one can attempt to develop a scheduling algorithm that is not restricted to single appearance schedules and attempts to jointly minimize the code size and buffer memory requirement. A further step in this scheduling problem is incorporating considerations that relate to buffer overlaying.

9.3 Parallel Computation

A number of scheduling techniques have been developed for compiling SDF graphs into efficient code for multiprocessor systems, for example [Sih91, Prin91, Liao93]. However these techniques do not consider code size constraints. Thus, it would be useful to extend the loose interdependence scheduling framework to address parallelism as well as memory requirements. An interesting problem that arises in this domain is the construction of optimal single appearance *parallel* schedules.

One restricted class of single appearance parallel schedules that would be useful to consider is that in which each schedule loop is either a serial loop, whose iterations are to be executed in succession as in the uniprocessor scheduling case, or a **doall** loop — a loop in which all iterations can execute simulta-

neously without any synchronization between them [Zima90]. Given a schedule loop in this model, it is executed serially if for some invocation I of the loop, data produced by some iteration of I is consumed by another iteration of I. If all iterations of a given invocation of a schedule loop are independent, then all iterations are executed in parallel. The execution "mode" of each loop can be represented by appending a letter s to the iteration count if it is to be executed serially and a letter p if it is to be executed as a doall loop. Thus, for example the schedule $(2p(2sA)(4pB))$ corresponds to the processor-time execution profile in Figure 9.2.

The basic problem to address in constructing this type of single appearance parallel schedule is determining the schedule that maximizes the throughput. Unlike the problem of minimizing code size, the solution to an instance of this problem depends in general on the execution time of each actor invocation, and thus a solution cannot be obtained from a topological analysis alone. This complication applies even to homogeneous SDF graphs. As a simple example, consider the homogeneous graph in Figure 9.3, and consider the single appearance parallel schedules,

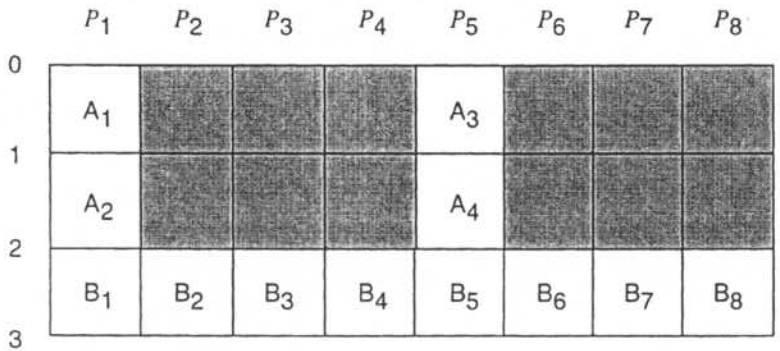


Figure 9.2. The processor-time execution profile for the single appearance parallel schedule $(2p(2sA)(4pB))$. The vertical axis corresponds to time and the horizontal co-ordinate identifies one of eight available processors $P_1 - P_8$. It is assumed that each actor invocation takes one time unit. A shaded region indicates that no operation is performed.

$$S_1 \equiv (6s(3s(5pA))(5s(3pB)))(90pC),$$

$$S_2 \equiv (5s(4s(5pA))(10s(2pB)))(100pC), \text{ and}$$

$$S_3 \equiv (8s(3s(4pA))(4s(3pB)))(96pC).$$

If we denote the execution times of actors A, B and C by t_A, t_B, t_C, respectively, then we can measure the throughput of S_1 as $throughput(S_1) = 90/(6(3t_A + 5t_B) + t_C)$ minimal schedule periods per unit time. Similar expressions can easily be derived for the throughput of S_2 and S_3. Table 9.1 lists the throughput of each schedule for three different sets of execution times.

Table 9.1 Throughput as a function of execution times

t_A	t_B	t_C	$throughput(S_1)$	$throughput(S_2)$	$throughput(S_3)$
500	150	1	0.00667	0.00571	0.00571
50	1	950	0.0479	0.0500	0.0400
1	950	950	0.00305	0.00206	0.00306

In this table, we see that each set of execution times corresponds to a different throughput-minimizing schedule from among the three schedules considered. Thus, we see that even for homogeneous SDF graphs, the construction of optimal single appearance parallel schedules (with regards to throughput) cannot be based solely on topological considerations.

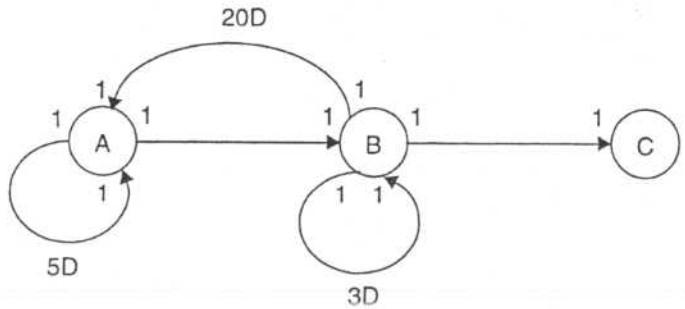


Figure 9.3. An example used to illustrate the problem of constructing single appearance parallel schedules.

When constructing single appearance parallel schedules or more general looped schedules for parallel computation, it would be useful to consider the time required for interprocessor communication. Scheduling techniques for SDF graphs that take interprocessor communication into account have been developed by Liao *et al.* [Liao93] and Sih [Sih91]; however these techniques do not attempt to construct loops in the target code.

BIBLIOGRAPHY

[AbuS81]
W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Transactions on Computers*, **Vol. C-30, No. 5**, May, 1981.

[Acke82]
W. B. Ackerman, "Data Flow Languages," *IEEE Computer Magazine*, **Vol. 15, No. 2**, February, 1982.

[Aho88]
A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[Alle87]
R. Allen and K. Kennedy, "Automatic Transformation of Fortran Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, **Vol. 9, No. 4**, October, 1987.

[Ambl92]
A. L. Ambler, M. M. Burnett, and B. A. Zimmerman, "Operational Versus Definitional: A Perspective on Programming Paradigms," *IEEE Computer Magazine*, **Vol. 25, No. 9**, September, 1992.

[Arvi90]
Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Computers*, **Vol. C-39, No. 3**, March, 1990.

[Ashc75]
E. A. Ashcroft, "Proving assertions about Parallel Programs," *Journal of Computer and Systems Science*, **Vol. 10, No. 1**, 1975.

[Bane88]
U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[Barr91]
B. Barrera and E. A. Lee, "Multirate Signal Processing in Comdisco's SPW," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Toronto, April, 1991.

[Bhat93]
S. Bhattacharyya and E. A. Lee, "Scheduling synchronous dataflow graphs for efficient looping", *Journal of VLSI Signal Processing*, **Vol. 6, No. 3**, December, 1993.

[Bhat94a]
S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms", *IEEE Transactions on Signal Processing*, **Vol. 42, No. 5**, May, 1994.

[Bhat94b]
S. S. Bhattacharyya, *Compiling Dataflow Graphs for Signal Processing*, Ph.D. thesis, Memorandum No. UCB/ERL M94/52, Electronics Research Laboratory, University of California at Berkeley, July, 1994.

[Bhat95]
S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams," *IEEE Workshop on VLSI Signal Processing*, Osaka, Japan, October 1995.

[Bier93]
J. C. Bier, P. D. Lapsley, and E. A. Lee, *Design Tools and Methodologies for DSP Systems — Volume 1: DSP Design Challenges, Methodologies, and Tools*, Berkeley Design Technologies, Inc., Fremont, California, 1993.

[Buck91]
J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing In Ptolemy," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Toronto, April, 1991.

[Buck93]
J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph.D. thesis, Memorandum No. UCB/ERL M93/69, Electronics Research Laboratory, University of California at Berkeley, September, 1993.

[Buck94]
J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, **Vol. 4**, April, 1994.

[Chas84]
M. Chase, "A Pipelined Dataflow Architecture for Signal Processing: the NEC μPD7281," *VLSI Signal Processing*, IEEE Press, 1988.

[Chow88]
F. C. Chow, "Minimizing Register Usage Penalty at Procedure Calls," *SIGPLAN Notices*, **Vol. 23, No. 7**, 1988.

[Corm90]
T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[Covi87]
C. D. Covington, G. E. Carter, and D. W. Summers, "Graphic Oriented Signal Processing Language — GOSPL," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April, 1987.

[Cubr93]
M. Cubric and P. Panangaden, "Minimal Memory Schedules for Dataflow Networks," *CONCUR '93*, Hildesheim, Germany, August, 1993.

[Davi89]
J. W. Davidson and D. B. Whalley, *Methods for Saving and Restoring Register Values Across Function Calls*, Technical Report 89-11, Department of Computer Science, University of Virginia, 1989.

[Davi92]
J. W. Davidson and A. M. Holler, "Subprogram Inlining: A Study of its Effects on Program, Execution Time," *IEEE Transactions on Software Engineering*, **Vol. 18, No. 2**, February, 1992.

[Denn75]
J. B. Dennis, *First Version of a Data Flow Procedure Language*, MAC Technical Memorandum 61, Laboratory for Computer Science, Massachusetts Institute of Technology, May, 1975.

[Denn80]
J. B. Dennis, "Dataflow Supercomputers," *IEEE Computer Magazine*, **Vol. 13, No. 11**, November 1980.

[Denn92]
J. B. Dennis, *Stream Data Types for Signal Processing*, Technical Report, September, 1992.

[Desm93]
D. Desmet and D. Genin, "ASSYNT: Efficient Assembly Code Generation for DSPs Starting from a Data Flowgraph," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, April, 1993.

[Dong79]
J.J. Dongarra and A.R. Hinds, "Unrolling Loops in FORTRAN," *Software-Practice and Experience*, **Vol. 9**, March, 1979.

[Egol93]
T. Egolf, S. Famorzadeh, and V. Madisetti, *On Library-Based Compiler Optimization for Programmable DSPs*, Technical Report, School of Electrical Engineering, Georgia Institute of Technology, December, 1993.

[Fish84]
J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer Magazine*, **Vol. 17, No. 7**, July, 1984.

[Gao92]
G. R. Gao, R. Govindarajan, and P. Panangaden, "Well-Behaved Programs for DSP Computation," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, March, 1992.

[Garb90]
J. Garbers, H. J. Promel, and A. Steger, "Finding Clusters in VLSI Circuits," *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, November, 1990.

[Gare79]
M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., NY, 1979

[Geni89]
D. Genin, J. De Moortel, D. Desmet, and E. Van de Velde, "System Design, Optimization, and Intelligent Code Generation for Standard Digital Signal Processors," *Proceedings of the International Symposium on Circuits and Systems*, Portland, Oregon, May, 1989.

[Geni90]
D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. De Man, "DSP Specification Using the Silage Language," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Gera92]
A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, **Vol. 16, No. 4**, December, 1992.

[Gurd85]
J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, **Vol. 28, No. 1**, January, 1985.

[Hart88]
J. Hartung, S. L. Gay, and S. G. Haigh, "A Practical C Language Compiler/Optimizer for Real Time Implementation on a Family of Floating Point DSPs," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, New York, April, 1988.

[Henn90]
J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[Ho88a]
W. H. Ho, *Code Generation for Digital Signal Processors Using Synchronous Dataflow*, Master's project report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1988.

[Ho88b]
W. H. Ho, E. A. Lee, and D. G. Messerschmitt, "High Level Dataflow Programming for Digital Signal Processing," *VLSI Signal Processing III*, IEEE Press, 1988.

[Hopc79]
J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 1979

[How90]
S. How, *Code Generation for Multirate DSP Systems in Gabriel*, Master's project report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1990.

[Huda89]
P. Hudak, "Introduction to Haskell and Functional Programming," *ACM Computing Surveys*, September, 1989

[Kafk90]
S. M. Kafka, "An Assembly Source Level Global Compactor For Digital Signal Processors," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Kapl87]
D. J. Kaplan, *et al.*, *Processing Graph Method Specification Version 1.0*, Unpublished memorandum, Naval Research Laboratory, Washington D.C, December, 1987.

[Karj88]
M. Karjalainen and S Helle, "Block Diagram Compilation and Graphical Editing of DSP Algorithms in the QuickSig System", *Proceedings of the International Symposium on Circuits and Systems*, Espoo, Finland, June, 1988.

[Karp66]
R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Math*, **Vol. 14, No. 6**, November, 1966.

[Kell61]
J. Kelly, Lochbaum, and V. Vyssotsky, "A Block Diagram Compiler," *Bell System Technical Journal*, **Vol. 40, No. 3**, May, 1961.

[Kern70]
B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, **Vol. 49, No.2**, February 1970.

[Kogg81]
P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw Hill, 1981.

[Lauw90]
R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine*, **Vol. 7, No. 2**, April, 1990.

[Lauw94]
R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, "Geometric Parallelism and Cyclo-Static Data Flow in GRAPE-II," presented at *IEEE Workshop on Rapid System Prototyping*, Grenoble, June, 1994.

[Lear90]
K. W. Leary and W. Waddington, "DSP/C: A Standard High Level Language for DSP and Numeric Processing," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Lee86]
E. A. Lee, *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*, Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1986.

[Lee87]
E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, **Vol. C-36, No. 2**, February, 1987.

[Lee88a]
E. A. Lee, "Recurrences, Iteration and Conditionals in Statically Scheduled Block Diagram Languages," *VLSI Signal Processing III*, IEEE Press, 1988.

[Lee88b]
E. A. Lee, "Programmable DSP Architectures: part I," *IEEE ASSP Magazine*, **Vol. 5, No. 4**, October, 1988.

[Lee89]
E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **Vol. 37, No. 11**, November, 1989.

[Lee91]
E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, **Vol. 2, No. 2**, April, 1991.

[Lee93]
E. A. Lee, "Multidimensional Streams Rooted in Dataflow," *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grained Parallelism*, Orlando, January, 1993.

[Lee95]
E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, **Vol. 83, No. 5**, May, 1995.

[Leis83]
C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Third Caltech Conference on VLSI*, March, 1983.

[Liao93]
G. Liao, G. R. Gao, E. Altman, and V. K. Agarwal, *A Comparative Study of DSP Multiprocessor List Scheduling Heuristics*, Technical Report, School of Computer Science, McGill University.

[Liao95]
S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Code Optimization Techniques for Embedded DSP Microprocessors," *Proceedings of the 32nd Design Automation Conference*, June, 1995.

[McGr83]
J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, *SISAL: Streams and Iteration in a Single Assignment Language: Language Reference Manual Version 1.1*, Lawrence Livermore Laboratory, July, 1983.

[Mess84]
D. G. Messerschmitt, "Structured Interconnection of Signal Processing Programs," *Proceedings of Globecom*, Atlanta, 1984.

[Miln90]
R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, 1990.

[Muel92]
F. Mueller and D. B. Whalley, "Avoiding Unconditional Jumps by Code Replication," *International Conference on Programming Language Design and Implementation*, San Francisco, June, 1992.

[Mura71]
Y. Muraoka, *Parallelism Exposure and Exploitation in Programs*, Ph.D. thesis, Report 71-424, Department of Computer Science, University of Illinois at Urbana-Champaign, February, 1971.

[Murt93]
P. K. Murthy, *Multiprocessor DSP Code Synthesis in Ptolemy*, Master's project report, Memorandum No. UCB/ERL M93/66, Electronics Research Laboratory, University of California at Berkeley, August, 1993.

[Murt94a]
P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, "Minimizing Memory Requirements for Chain Structured Synchronous Dataflow Graphs", *Proceedings of the ICASSP '94*, Adelaide, Australia, April, 1994.

[Murt94b]
P. K. Murthy and E. A. Lee, "On the Optimal Blocking Factor for Blocked, Non-Overlapped Schedules", *Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, November, 1994.

[Murt94c]
P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, "Combined Code and Data Minimization for Synchronous Dataflow Programs", ERL Memo No. UCB/ERL M94/93, Electronics Research Lab, UC Berkeley, CA 94720.

[Najj92]
W. A. Najjar, R. Roh, and A. P. Wim Bohm, "Initial Performance of a Bottom-Up Clustering Algorithm for Dataflow Graphs," *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, January, 1993.

[Naye93]
K. Nayebi, T. P. Barnwell, and M. J. T. Smith, "Nonuniform Filter Banks: A Reconstruction and Design Theory," *Transactions on Signal Processing*, **Vol. 41, No. 3**, March, 1993.

[Ohal91]
D. R. O'Hallaron, *The Assign Parallel Program Generator*, Memorandum CMU-CS-91-141, School of Computer Science, Carnegie Mellon University, May, 1991.

[Olso92]
T. J. Olson, N. G. Klop, M. R. Hyett, and S. M. Carnell, "MAVIS: a Visual Environment for Active Computer Vision," *Proceedings of the 1992 IEEE Workshop on Visual Languages*, Seattle, September, 1992.

[Pino94]
J. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," invited paper in *Journal of VLSI Signal Processing*, to appear in 1994.

[Powe92]
D. B. Powell, E. A. Lee, and W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, March, 1992.

[Prin91]
H. Printz, *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*, Ph.D. thesis, Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, May, 1991.

[Prin92]
H. Printz, "Compilation of Narrowband Spectral Detection Systems for Linear MIMD Machines," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August, 1992.

[Rasu94]
K. Konstantinides and J. R. Rasure, "The Khoros Software Development Environment for Image and Signal Processing," *IEEE Transactions on Image Processing*, **Vol. 3, No. 3**, May, 1994.

[Reek92]
H. J. Reekie, "Integrating Block-Diagram and Textual Programming for Parallel DSP," *Proceedings of the 3d International Symposium on Signal Processing and its Applications*, Queensland, Australia, August, 1992.

[Reit68]
R. Reiter, "Scheduling Parallel Computations," *Journal of the ACM*, **Vol. 15, No. 4**, October, 1968.

[Ritz92]
S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August, 1992.

[Ritz93]
S. Ritz, M. Pankert, and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," *Proceedings of the International Conference on Application-Specific Array Processors*, Venice, October, 1993.

[Ritz95]
S. Ritz, M. Willems, H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," *Proceedings of the ICASSP 95*, Detroit, Michigan, May 1995.

[Schm91]
U. Schmidt and K. Caesar, "Datawave: a Single-Chip Multiprocessor for Video Applications," *IEEE Micro Magazine*, **Vol. 11, No. 3**, June, 1991.

[Shan87]
K. S. Shanmugan, G. J. Minden, E. Komp, T. C. Manning, and E. R. Wiswell, *Block-Oriented System Simulator (BOSS)*, Telecommunications Laboratory, University of Kansas, Internal Memorandum, 1987.

[Sih91]
G. C. Sih, *Multiprocessor Scheduling to Account for Interprocessor Communication*, Ph.D. thesis, Memorandum No. UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, April, 1991.

[Tarj72]
R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal of Computing*, June, 1972.

[Tow88]
J. Tow, S. L. Gay, and J. Hartung, "Implementation of DSP Applications Using the AT&T DSP32C Compiler and Application Library," *Proceedings of the International Symposium on Circuits and Systems*, Espoo, Finland, June, 1988.

[Vaid93]
P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[Veig90]
M. Veiga, J. Parera, and J. Santos, "Programming DSP Systems on Multiprocessor Architectures," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April, 1990.

[Wolf89]
M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

[Wolf91]
M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Francisco, June, 1991.

[Yu93]
K. H. Yu and Y. H. Hu, "Optimized Code Generation for Programmable Digital Signal Processors," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, April, 1993

[Zima90]
H.Zima and B.Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1990.

[Ziss87]
M. A. Zissman, G. C. O'Leary, and D. H. Johnson, "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer," *Proceedings of*

the International Conference on Acoustics, Speech, and Signal Processing, Dallas, April, 1987.

[Zivo93]
V. Zivojnovic, S. Ritz, and H. Meyr, "Multirate Retiming: A Powerful Tool for Hardware/Software Codesign," Technical Report, Institute for Integrated Systems in Signal Processing, Aachen University of Technology, 1993.

[Zivo95]
V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen, "DSPs, GPPs, and Multimedia Applications — An Evaluation Using DSPStone," *Proceedings of ICSPAT*, November, 1995.

Index

$\lfloor\ \rfloor$ 29
$\lceil\ \rceil$ 29
$|\,|$ 29
$-$ 29
$\langle\rangle$ 31
$\rightarrow$ 31
η 123
O 33
Θ 33
ρ 65
ρ_G 65
Ω 33

A
actor 37
actors 58
acyclic 31
acyclic precedence graph 46
acyclic scheduling algorithm 95
adjacent pair 30
adjacent vertices 30
admissable schedule 38
admissible firing 38
ancestor 31
ancs 31
APG 46
APGAN 142
APGAN candidate 146
APGAN instance 146
appearances 58
associated graph of a path 31
atomic dataflow 9

B
balance equations 40
blocked schedule 24
blocking factor 24, 44
blocking vector 64
BMLB 123
BMLB Schedule 123
BMUB 125
body of a schedule loop 57
boundary vertex 136
bounding constant. RPMC 135
buffer 37
buffer memory 60
buffer memory requirement 60
buffer memory, shared 61
buffer_memory 60
buffering 37

C
chain-structured graph 33
class-S algorithms 49
clust 67
clusterable 67
clustering into 66
cns 37
computation graph 10, 91
ComputeRepetitions algorithm 48
connected component 32
connected component subgraph 33
consistent SDF graph 45
ConstructValidSchedule algorithm 49
contained in 58
coprime 29, 43, 82
corresponds to (edges) 67
cut 135
cycle 31

D
dataflow 6
deferrable actor 54
del 37
delay 37
delayless version 38
denom 29
desc 31
descendent 31
directed cycle 31
directed multigraph 30
doall loop 171
DPPO 126

E
edge 30
edges 37
Euclid's algorithm 43
example, modem 103
example, non-uniform filterbank 139
example, sample-rate conversion 11, 62, 131
example, satellite receiver 159
example, tree-structured filterbank 159
executing a schedule 38
exponential time algorithm 34

F
factoring a schedule loop 70
FIFO 37
fine grain dataflow 9
fireable 38
firing 37
flat single appearance schedule 85
fully reduced 82
fundamental cycle 31

G
gcd 29
GDPPO 127

H
homogeneous SDF graph 37

I
independent 31
independent 31
input buffering 132
input edge 30
introduces a cycle 67
inv 39, 58
invocation 38
invocation number 38
invocation sequence generated by a schedule 58
isomorphic SDF graphs 47
iterand of a looped schedule 57
iterand of a schedule loop 57
iteration count 57
iteration period bound 25

J
J 64
J 44

L
latency 132
lcm 29
least common multiple 43
legal cut, RPMC 135
lexical ordering 59
lexorder 59
loop 57
loop fusion 72
looped schedule 57
loose interdependence algorithms 95
loosely interdependent 94

M
max 29
max_tokens 60
min 29
minimal periodic schedule 44
minimum buffer scheduling 49
mixed grain dataflow 9

N
nested in 58
nonR 120
NP-completeness 35
null schedule 38
null schedule loop 57
null space 43
numer 29

O
one-iteration loop 58
order-optimal 125
output edge 30

P
P 59
part of a subschedule 59
path 31
periodic schedule 39
polynomial time algorithm 34
polynomial time reducible 35
position 59
prd 37

preamble 92
predecessor 30
projection 63
projection 63
proper clustering condition 147

Q
q 43
q_G 65
$\mathbf{q}_{R/G}$ 65

R
random schedule 161
rate-optimal schedule 25
R-condition 119
reduced fraction 43
ReducedFraction 29
repetition count 65
repetitions vector 43
repetitions vector of a subgraph 65
retiming 170
root strongly connected component 33
root vertex 33
RPMC 134
R-schedule 120

S
sample rate consistent 45
schedule 38
schedule loop 57
schedule period 38
SDF graph 37
self-loop 40
single appearance schedule 58
sink vertex 30
S_L 95
snk 30
source vertex 30
spanning tree 42
src 30
state 38
static scheduling 12
strongly connected 32
strongly connected component 32
strongly connected component subgraph 33
strongly connected components algorithm 95
subgraph 30
subgraph 31
subindependence 87
subindependence partitioning algorithm 95
subindependent partition 87
subroutine vs inline code 17
subschedule 57
successor 30
synthesis (compilation model) 16

T
T 59
termination of a schedule 39
threading (compilation model) 16
tight scheduling algorithm 95
tightly interdependent 94
tightly interdependent component 100
TNSE 47
token 37
topological sort 33
topology matrix 40
transitive edge 31
tree-structured graph 41
trivial directed multigraph 30

V
valid schedule 45
vectorization 113
vertex 30

W
well-ordered graph 33