
**A CODE MAPPING SCHEME
FOR DATAFLOW SOFTWARE
PIPELINING**

**THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE**

**PARALLEL PROCESSING AND FIFTH
GENERATION COMPUTING**

Consulting Editor

Doug DeGroot

Other books in the series:

PARALLEL EXECUTION OF LOGIC PROGRAMS, J. S. Conery
ISBN: 0-89838-194-0

**PARALLEL COMPUTATION AND COMPUTERS FOR ARTIFICIAL
INTELLIGENCE, J. S. Kowalik**
ISBN: 0-89838-227-0

MEMORY STORAGE PATTERNS IN PARALLEL PROCESSING, M. E. Mace
ISBN: 0-89838-239-4

SUPERCOMPUTER ARCHITECTURE, P. B. Schneck
ISBN: 0-89838-234-4

**ASSIGNMENT PROBLEMS IN PARALLEL AND DISTRIBUTED
COMPUTING, S. H. Bokhari**
ISBN: 0-89838-240-8

MEMORY PERFORMANCE OF PROLOG ARCHITECTURES, E. Tick
ISBN: 0-89838-254-8

DATABASE MACHINES AND KNOWLEDGE BASE MACHINES,
M. Kitsuregawa
ISBN: 0-89838-257-2

PARALLEL PROGRAMMING AND COMPILERS, C. D. Polychronopoulos
ISBN: 0-89838-288-2

A HIGH PERFORMANCE ARCHITECTURE FOR PROLOG, T. P. Dobry
ISBN 0-8923-9060-1

**PARALLEL MACHINES. PARALLEL MACHINE LANGUAGE: The Emer-
gence of Hybrid Dataflow Computer Architectures, R.A. Iannucci**
ISBN 0-7923-9101-2

A CODE MAPPING SCHEME FOR DATAFLOW SOFTWARE PIPELINING

by

Dr. Guang R. Gao

McGill University
School of Computer Science

Foreword by

Jack B. Dennis

Massachusetts Institute of Technology



SPRINGER-SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Gao, Guang R.

A code mapping scheme for dataflow software pipelining / Guang R. Gao ; foreword by Jack B. Dennis.

p. cm. — (The Kluwer international series in engineering and computer science ; 0125. Parallel processing and fifth generation computing)
Revision of thesis (Ph. D.)—Massachusetts Institute of Technology, 1986.
Includes bibliographical references and index.

ISBN 978-1-4613-6782-6 ISBN 978-1-4615-3988-9 (eBook)

DOI 10.1007/978-1-4615-3988-9

1. Computer architecture. 2. Parallel processing (Electronic computers)
3. Computer software. I. Title II. Series: Kluwer international series in engineering and computer science ; SECS 125. III. Series: Kluwer international series in engineering and computer science. Parallel processing and fifth generation computing.

QA76.9.A73G36 1990

004.2'2—dc20

90-49763

CIP

Copyright 1991 by Springer Science+Business Media New York

Originally published by Kluwer Academic Publishers in 1991

Softcover reprint of the hardcover 1st edition 1991

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, Springer-Science +Business Media, LLC.

Printed on acid-free paper.

Contents

List of figures	x
Preface	xv
Acknowledgements	xvii
Foreword	xix
1 Introduction	1
1.1 Array Operations in Numerical Computation	3
1.2 Vector Processing and Vectorizing Compilers	4
1.3 Dataflow Computers	6
1.4 Granularity and Functionality	8
1.5 A Pipelined Code Mapping Scheme	9
1.5.1 Fine-Grain Parallelism and Pipelining of Data Flow Programs	9
1.5.2 Data Flow Languages and Array Computations . .	11
1.5.3 Dataflow Software Pipelining	13
1.5.4 Pragmatic Aspects of Compiler Construction . . .	14
1.6 Synopsis	15
2 The Static Data Flow Model	17
2.1 Static Data Flow Graph Model	18
2.1.1 The Basic Model	18
2.1.2 The Determinancy and Functionality of Static Data Flow Graphs	25
2.1.3 Static Data Flow Computers	26
2.1.4 Terminologies and Notations	27

2.2	Pipelining of Data Flow Programs	28
2.2.1	Basic Concepts of Pipelining	29
2.2.2	Timing Considerations	29
2.2.3	Throughput of Pipelining Data Flow Programs . .	32
2.3	Balancing of Data Flow Graphs	34
2.4	Pragmatic Issues Regarding the Machine Model and Balancing	38
3	Algorithmic Aspects of Pipeline Balancing	41
3.1	Weighted Data Flow Graphs	43
3.2	Related Work on Balancing Techniques	44
3.2.1	Balancing Techniques	44
3.2.2	Relationship between Balancing and Optimization	47
3.3	A Linear Programming Formulation of the Optimization Problem	50
3.4	Solution of Optimal Balancing Problems	54
3.4.1	An Example	54
3.4.2	Solution Techniques	56
3.5	Extensions to the Results	56
3.5.1	Graphs with Multiple Input and Output Nodes . .	56
3.5.2	Conditional Subgraphs	58
4	Source Program Structure and Notation	61
4.1	The PIPVAL Language	62
4.1.1	Overview	62
4.2	Array Operation Constructs	67
4.2.1	FORALL Expressions	68
4.2.2	FOR-CONSTRUCT Expressions	70
4.3	Code Blocks with Simple Nested Structure	72
5	Basic Pipelined Code Mapping Schemes	77
5.1	Data Flow Representation of Arrays	78
5.2	Basic Mapping Schemes	80
5.3	SDFGL—A Static Data Flow Graph Language	81
5.3.1	The Basic Language	81
5.3.2	An Example of SDFGL Graph Representation . .	83
5.3.3	Extension of the Basic SDFGL	83

6	Mapping Rules for Expressions without Array Creation Constructs	89
6.1	M[id], M[const], M[op exp], and M[exp op exp] Mapping Rules	91
6.2	Mapping Rule for exp,exp	91
6.3	Mapping Rule for LET-IN Expressions	92
6.4	Mapping Rule for Conditional Expressions	93
6.4.1	A Simple Conditional Expression	93
6.4.2	Mapping of Conditional Expressions with Multiple Arms	94
6.5	Mapping Rule for Array Selection Operations	102
6.6	Pipelining of Graphs for Simple Primitive Expressions . .	102
6.7	An Overview of Mapping General Iteration Expressions .	106
7	Mapping Scheme for One-Level FORALL Expressions	109
7.1	Basic Mapping Rule	109
7.1.1	Pipelined Mapping Strategy	109
7.1.2	Basic Mapping Rule	112
7.2	Optimization of Array Operations	117
7.2.1	An Example	118
7.2.2	Array Selection Operations Having Compile-Time Computable Selection Index Ranges	120
7.2.3	Selection Index Ranges Which Cannot Be Computed at Compile-Time	130
7.3	Pipelining One-Level Primitive FORALL Expressions . .	130
8	Mapping Scheme for Multi-Level FORALL Expressions	133
8.1	Representation of Multi-Dimensional Arrays	133
8.1.1	Flattened Representation	133
8.1.2	Index Vectors and Their Orders	135
8.1.3	Major Orders in Flattened Data Flow Representation	136
8.2	Mapping Two-Level FORALL Expressions	137
8.3	The Optimization of Two-Level Primitive FORALL Expressions	141
8.3.1	Consistent Array Selection Orders	141
8.3.2	An Example	143
8.3.3	Optimization of Array Selection Operations Having Compile-Time Computable Index Ranges . . .	145
8.4	Multi-Level FORALL Expressions	150

9	Mapping Scheme for FOR-CONSTRUCT Expressions	155
9.1	The Basic Mapping Rule	155
9.2	Optimization Procedures	163
9.3	An Example Optimization	166
10	Related Optimization Techniques	171
10.1	Using Companion Pipelines to Solve Linear Recurrences .	172
10.1.1	Mapping of First-Order Linear Recurrences	172
10.1.2	A Second-Order Linear Recurrence	178
10.1.3	Discussion	182
10.2	Enhancing Pipelining by Loop Unfolding and Interchanging	182
10.2.1	Loop Unfolding	182
10.2.2	Interchanging the Order of the Innermost Loop . .	185
10.2.3	A Matrix Multiplication Example	187
10.3	Multiple Pipelines	189
11	Program Structure, Compilation, and Machine Design	193
11.1	Overview of Program Structure Analysis	193
11.2	Considerations for Analyzing a Certain Class of Programs	195
11.2.1	A Cluster Entirely of Primitive FORALL Blocks .	196
11.2.2	Nodes for Primitive FOR-CONSTRUCT Expressions	198
11.2.3	Remarks	200
11.3	Pragmatic Compiler Construction	200
11.4	Considerations in Instruction Set Design	202
11.4.1	Instructions for Conditionals	202
11.4.2	Control Value Sequences	203
11.4.3	The IGEN Instruction	203
11.5	Machine Support for Array Operations	204
11.5.1	Flattening of Arrays	204
11.5.2	Flattening and Pipelining	205
11.5.3	Array Memory Management in a Target Machine .	205
11.6	FIFO Implementation	207
12	Efficient Dataflow Software Pipelining	211
12.1	Overview	211
12.2	An Argument-Fetching Dataflow Architecture	213
12.2.1	Architectural Model	213
12.2.2	Data-Driven Program Tuples—A-Code	214
12.3	A Compiler Prototype	216

12.3.1	The Testbed	216
12.3.2	Performance Metrics	217
12.4	Performance Analysis	219
12.4.1	Pipelining of the Livermore Loops	219
12.4.2	Optimization by Balancing	221
12.5	Impacts of Architecture Configuration	223
12.5.1	Architectural Factor I: Signal Processing Capacity	223
12.5.2	Architectural Factor II: The Enabled Instruction Scheduler	226
12.6	Related Work and Discussion	228
13	Conclusions	231
13.1	Summary	231
13.2	Topics for Future Research	232
	Bibliography	235
	Index	247

List of Figures

1.1	A Static Dataflow Architecture	7
1.2	Pipelining of Data Flow Programs	10
1.3	Pipelining of Array Operations	14
1.4	A Group of Code Blocks	15
2.1	A Static Data Flow Graph	19
2.2	Firing Rules for T-gate and F-gate Actors	20
2.3	Firing Rules for Merge Actors	21
2.4	Firing Rules for Switch Actors	22
2.5	Implementation of Switch Actors Using T-gate and F-gate Actors	23
2.6	A Conditional Subgraph	23
2.7	An Iteration Subgraph	24
2.8	Acknowledgement Arcs	26
2.9	An Example of Pipelining	30
2.10	One Run of a Data Flow Graph	32
2.11	A Data Flow Graph with Maximum Throughput of $1/4 \tau$	35
2.12	A Balanced Data Flow Graph	35
2.13	Stage Partitioning of a Balanced Pipeline	37
3.1	An Example of Balancing	42
3.2	A Weighted Data Flow Graph	44
3.3	A Topologically Sorted Graph	46
3.4	An Example of Applying Algorithm 4.1	48
3.5	Problems with the Classical Balancing Approach	49
3.6	Delay Changes Due to Buffer Moving	51
3.7	Earliest and Latest Firing Times	52
3.8	A Matrix Notation for Example 3.1	55
3.9	Balancing of Conditional Subgraph	58
4.1	The Syntax of PIPVAL	64

4.2	An Example of a Two-Level forall Expression	70
4.3	An Example of a Val for-iter Construct for Creating an Array	71
4.4	A Two-Level Primitive for-construct Expression	73
4.5	A Two-Level Mixed Code Block, example 1	74
4.6	A Two-Level Mixed Code Block, example 2	75
5.1	Data Flow Representations of a One-Dimensional Array	78
5.2	Major Orders for a Flattened One-Dimensional Array	80
5.3	An Example of SDFGL, part 1	84
5.4	An Example of SDFGL, part 2	85
5.5	An Example of Named Subgraphs in SDFGL	86
6.1	Mapping Rules for $M[id]$, $M[const]$, and $M[op\ exp]$	90
6.2	Mapping Rule for $M[exp_1\ op\ exp_2]$	92
6.3	Mapping Rule for exp, exp	93
6.4	Mapping Rule for a let-in Expression	94
6.5	The SDFGL Graph for Mapping a let-in Expression	95
6.6	Mapping Rule for Simple Conditional Expressions	96
6.7	The SDFGL Graph of a Mapped Simple Conditional Expression	97
6.8	A Multi-Armed Conditional Expression	98
6.9	The SDFGL Graph of a Multi-Armed Conditional Expression, version 1	99
6.10	A SDFGL Graph for a Multi-Armed Conditional Expression, version 2	100
6.11	The MB Subgraph	101
6.12	Mapping Rule for a Multi-Armed Conditional Expression	103
6.13	SDFGL Graph Showing the Mapping of a Multi-Armed Conditional Expression	104
6.14	Mapping Rule for an Array Selection Operation	105
6.15	Mapping of a for-iter Expression	107
7.1	Parallel and Pipelined Mapping Schemes for a forall Expression	111
7.2	Mapping Rule for a forall Expression	114
7.3	The SDFGL Graph for Mapping a forall Expression	115
7.4	A Simplified Version of Figure 7.3	116
7.5	The SDFGL Graph for Mapping a One-Level forall Expression	117

7.6	The Result after Removing Array Operations in Figure 7.5	119
7.7	Selection Index Ranges for Array Operations in Figure 7.5	119
7.8	A Range-Partitioning Conditional Expression	120
7.9	The Optimization of a SEL Actor	122
7.10	Skews in the Optimization of Array Selection Operations	123
7.11	The Control Sequence of an MM Actor	124
7.12	Optimization Procedure for a Case-1 forall Expression . .	125
7.13	Optimization of AGEN	126
7.14	An Example of Optimization by OPMAP, after Step 2 . . .	127
7.15	Continued from Figure 7.14, after Step 3	128
7.16	Continued from Figure 7.15, after Step 5	129
7.17	Continued from Figure 7.16, Result	129
7.18	The RGEN Subgraph for Computing Control Sequence Pa- rameters	131
7.19	Optimization of a SEL Actor Using CGEN Subgraph	131
8.1	Flattened Data Flow Representations of a Two-Dimensional Array	134
8.2	A Pipelined Representation of a Sequence of Index Vectors	137
8.3	Orders between Input and Outputs of an Affine Function	138
8.4	The Model Problem, Version 1	138
8.5	The SDFGL Graph for Mapping a Two-Level forall Ex- pression	140
8.6	The Data Flow Graph for Mapping $A[i, j]$	141
8.7	An Array Selection Operation in a Two-Level Primitive forall Expression	142
8.8	The Selection Order of an Array Selection Operation . . .	143
8.9	The Selection Index Ranges and Control Sequences, Level- 1 SEL Actors	144
8.10	The Selection Index Ranges and Control Sequence, Level- 2 SEL Actors	145
8.11	Result of Removing Array Actors in Figure 8.5	146
8.12	Form for Standard Case 1	147
8.13	Optimization of a Level-1 SEL Actor	148
8.14	Skews in the Optimization with Weighted Buffer Size . . .	148
8.15	The Optimization Procedure for Two-Level Case-1 forall Expressions	149
8.16	A Two-Level forall Optimization Example, after Step 2 .	151
8.17	Continued from Figure 8.16, after Step 3	152
8.18	Continued from Figure 8.17, after Step 5	153

8.19	Continued from Figure 8.18, Final Result	154
9.1	The Data Flow Graph of a for-construct Expression . .	157
9.2	The Functions of IGEN and AGEN Subgraphs	158
9.3	Basic Mapping Rule for for-construct Expressions . . .	160
9.4	The SDFGL Graph from the Mapping of a for-construct Expression	161
9.5	The Data Flow Graph from Mapping a First-Order Linear Recurrence	162
9.6	A First-Order Linear Recurrence	162
9.7	An Example of a Two-Level for-construct Expression . .	163
9.8	The Mapping of a Two-Level for-construct Expression .	164
9.9	Optimization of AGEN in a for-construct Expression . . .	166
9.10	Selection Index Ranges and Control Sequences, Level-1 SEL Actors	167
9.11	Selection Index Ranges and Control Sequences, Level-2 SEL Actors	167
9.12	A Two-Level for-construct Optimization Example, after Removing Array Actors	168
9.13	A Two-Level for-construct Optimization Example, Fi- nal Result	169
10.1	The Data Flow Graph of a First-Order Linear Recurrence	173
10.2	A First-Order Linear Recurrence with Backup	175
10.3	A Maximally Pipelined Data Flow Graph for a FLR . . .	175
10.4	The Companion Pipeline in Figure 10.3	176
10.5	Pipelined Execution of FLR—The First Few Steps	177
10.6	A for-construct Expression for (10.5)	178
10.7	A Data Flow Graph for the Fibonacci Recurrence	179
10.8	The Fibonacci Recurrence after Transformation	180
10.9	The Data Flow Graph for the Transformed Fibonacci Re- currence	180
10.10	Pipelined Execution of the Transformed Fibonacci Recur- rence	181
10.11	Loop Unfolding	184
10.12	A Completely Unfolded Loop	185
10.13	Maximum Pipelining of a Completely Unfolded Loop . . .	186
10.14	Matrix-Vector Multiplication by Loop Interchanging and Unfolding	190
10.15	A Multiple Pipelined Mapping Scheme	190

11.1 An Acyclic Code Block Cluster, Example 1	197
11.2 An Acyclic Code Block Cluster, Example 2	198
11.3 An Acyclic Code Block Cluster, Example 3	199
11.4 Structure of a Dataflow Compiler	208
11.5 Flattening of an Array, a Selection Operation	208
11.6 Flattening of an Array, an Array Generation Subgraph	209
11.7 Using Flattening Techniques for Pipelining	209
12.1 An Argument-Fetching Dataflow Processor	214
12.2 A Dataflow Program Tuple	215
12.3 Testbed for the Prototype Compiler	216
12.4 The Macrosimulator	218
12.5 Varying Machine Configurations for loop7.	224
12.6 The Enabled Instruction Scheduler Bottleneck	227

Preface

This monograph evolved from my Ph.D dissertation completed at the Laboratory of Computer Science, MIT, during the Summer of 1986. In my dissertation I proposed a pipelined code mapping scheme for array operations on static dataflow architectures. The main addition to this work is found in Chapter 12, reflecting new research results developed during the last three years since I joined McGill University—results based upon the principles in my dissertation. The terminology *dataflow software pipelining* has been consistently used since publication of our 1988 paper on the *argument-fetching* dataflow architecture model at McGill University [43].

In the first part of this book we describe the static data flow graph model as an operational model for concurrent computation. We look at timing considerations for program graph execution on an ideal static dataflow computer, examine the notion of pipelining, and characterize its performance. We discuss balancing techniques used to transform certain graphs into fully pipelined data flow graphs. In particular, we show how optimal balancing of an acyclic data flow graph can be formulated as a linear programming problem for which an optimal solution exists. As a major result, we show the optimal balancing problem of acyclic data flow graphs is reduceable to a class of linear programming problem, the network flow problem, for which well-known efficient algorithms exist. This result disproves the conjecture that such problems are computationally hard.

The second part of the book concentrates on the development of a pipelined code mapping scheme for static dataflow computers. The key to our scheme is the pipelined mapping of array operations. After source and object languages are defined, our basic pipelined code mapping scheme is formulated, and the optimization of array operations is presented, each in an algorithmic fashion. The major result here is that a class of program blocks (expressible in **forall** or **for-construct**

expressions) can be effectively mapped into pipelined data flow graphs, including blocks having conditional and nested structures like those frequently encountered in numerical computation. Our mapping technique uses both global and local optimization, unified by our pipeline principle. Our treatment of array operations is unique in the sense that information about overall program structure guides code generation, allowing the massive parallelism within array operations to be exploited by the architecture in a fine-grain manner. Although the second part of the book concentrates on the formulation of a pipelined code mapping scheme, other related optimization techniques are described which improve the performance of data flow graphs.

The next part of the book addresses issues which are extensions to our work. One important extension is the construction of a compiler based upon pipelined code mapping. A discussion of the structure of application programs (the program block graphs shown in Figure 1.4) and their relationship to pipelined mapping schemes can be found in the first half of Chapter 11. There we outline the structure of a possible compiler which incorporates the principles developed from our research. Much interesting work remains to be done in this area, and our discussion suggests topics for further research. Another aspect, discussed in the last part of Chapter 11, is the impact of pipelined code mapping on the machine design.

The final part of the book is devoted to the analysis of the compiling schemes described in this monograph. We investigate the effects of software pipelining using realistic models having finite, as opposed to infinite, resources. Our target architecture is the McGill Dataflow Architecture (MDFA) which employs a conventional pipelined architecture to achieve pipelined instruction execution and a data-driven instruction scheduling mechanism to exploit fine-grain parallelism. Unlike many other dataflow architectures, the instruction execution phase of the MDFA is comparable to conventional von Neumann architectures, and the mechanism for fine-grain synchronization and scheduling is separate from the processing element, facilitating the study of compiler/architecture impacts on the fine-grain parallelism and avoiding the peculiarities of processing elements of particular architectures.

Although this book is based on the static dataflow model, I have long been convinced that our work could be extended to other dataflow models. In this regard, I am pleased that dataflow software pipelining, combined with VLIW scheduling techniques, has also been recently used in static loop scheduling for dynamic dataflow machines [23].

Acknowledgements

This book evolved from my graduate work at MIT between 1980–1986. Its preparation would not have been possible without the unique research and academic environment at MIT, in particular, the intellectually exciting and rewarding experience of being a part of the Computational Structures Group in the Laboratory of Computer Science.

I am most grateful to Jack Dennis, my thesis supervisor, for his patience, encouragement, and support. He introduced me to data flow at our first meeting ten years ago and arranged for me to participate in the Summer workshop on dataflow models held at the MIT Endicott estate, allowing me to meet many interesting researchers in the field just two weeks after I arrived. Throughout the years of my thesis research, his guidance was always present, shaping all aspects of my work. Besides suggesting many of my graduate courses, he encouraged me during my first semester at MIT to register for 6.001, the sophomore-level introductory course offered by the EECS Department and taught by H. Abelson and J. Sussman—an experience I found most rewarding.

I am also grateful to Arvind, who has given me constant support during my career in dataflow research and has patiently answered my many questions throughout the years. Likewise, I am grateful to John Guttag for his careful reading of my thesis and for many interesting suggestions for improving it.

I found the friendly and stimulating atmosphere in the Computation Structures Group at MIT to be a constant source of support. And for their support, I would like to thank the many members of the group: Rishiyur Nikhil, Clement Leung, Randy Bryant, Bill Ackerman, Dean Brock, Andy Boughton, Tan-Anh Chu, Willie Lim, Robert Iannucci, Greg Papadopoulos, Suresh Jagannathan, Keshav Pingali, David Culler, Kevin Theobald and others.

I would also like to thank the members of the Advanced Architecture and Program Structures Group of School of Computer Science at

McGill University for joining me in the adventure of dataflow research. In particular, I thank Herbert Hum, René Tio, Robert Yates, Zaharias Paraskevas, Yue-Bong Wong, and Jean Marc Monti for working on the architecture, compiler, and simulation testbed for the McGill Dataflow Architecture. Without their support, the work described in Chapter 12 would not have been possible.

I owe thanks to Jean Marc Monti and Russell Olsen for their assistance in the preparation of this manuscript, as well as for their many suggestions for its improvement.

Finally, I am grateful beyond words to my wife Ping and my son Ning, for their patience, support, companionship, and love. Ping typed the original manuscript of my thesis, spending many long nighttime hours.

Foreword

Parallel computation is now a major theme of computer science research, multiprocessor computers are becoming practical tools for scientific computation, and the use of computer systems executing myriads of concurrent transactions over local and long distance data networks is widespread in commerce. The importance of parallel computing stands in contrast with the dismal state of software technology for parallel computation: There is no generally accepted practical programming language or methodology for expressing programs for parallel computation; and there is no portability of programs between different models of parallel computers. Most present applications of parallel computation use operating system facilities for the coordination of concurrent activities, because practical programming languages offer no way of expressing programs to allow for parallel execution, but merely bring crude operating system mechanisms into the language.

In contrast to current practice in sequential programming, there is a chasm between the expectations for parallel computation on one hand, and the achievable performance and attractiveness of the programming environment on the other.

Then why is parallel processing so popular? The explanation lies in the amazing reduction in the size and cost of computing hardware over the past two decades, the absence of a matching increase in the speed of hardware components, and in the increasing appetite of commerce for networks of computers for transaction processing and information management in distributed organizations.

At MIT our work on computer system architecture starting in the 1960s has taken the view that computer architecture should strive to simplify the construction of software. The goal in our development of dataflow concepts was to provide a program execution model compatible with the sound principles of structured programming and language design that were then gaining recognition. The resulting formalism has

a strong kinship to the ideas of functional programming. Both dataflow and functional programming view a computational module as representing a mathematical function where the absence of side effects improves understandability and makes it easy to tell which parts may be executed concurrently.

The benefits of these developments will soon be realized in the area of large-scale scientific computation, where there is less to be lost in departing from convention and more to be gained in performance and programmer productivity. The work Dr. Gao reports in this volume is playing a major role in this forthcoming revolution in scientific computation. It recognizes the prominent role played by arrays in high performance numerical computation, and develops tools for implementing computations on arrays with high performance within the framework of dataflow computer architecture and functional programming languages. It is an important contribution to bridging the chasm of programmability and performance for parallel computers.

At the time Gao joined the MIT dataflow research team in 1980, we had just taken up the challenge of applying dataflow ideas to real world problems of scientific computation in a collaboration with the Lawrence Livermore National Laboratory. The design of the Val programming language had been completed to provide a functional language tool for writing numerical codes for analysis to determine the performance to be expected from a dataflow computer. An implementation at MIT by Ackerman and Brock had just become available. Upon his arrival at MIT, Gao became immediately involved in analyzing an important NASA benchmark program as expressed in the Val language. This is remarkable because Gao had left mainland China only a short time earlier and was just beginning to be comfortable with English.

The aspect of parallel computation that became Gao's thesis research is the structure of dataflow machine code that would lead to efficient implementation of the loops contained in important scientific codes. These loops often construct array values, and are of two basic kinds: parallel and sequential. In a parallel loop that constructs an array, each element is defined independently of all others in the array, and all elements may be computed in parallel. In a sequential loop there are dependences—each array element is defined in terms of elements defined in previous cycles of the loop. These correspond to mathematical recurrences.

The first form had been incorporated into the design of Val as the "forall" expression. Gao's work shows how to set up "software pipelines" that support efficient execution of these computations on dataflow com-

puters. He also shows how recurrences may be efficiently implemented and introduces a new kind of array constructor expression that provides a natural way of writing such array definitions in the functional programming style.

The work in this book will help bring the benefits of functional programming into the practice of scientific computing for the first time, an event that could mark the beginning of revolutionary changes in the way programs are built and the machines that run them are organized.

Jack B. Dennis

M.I.T.

Cambridge, Massachusetts