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 Emergence 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

Originally published by Kluwer Academic Publishers in 1991
Softcover reprint of the hardcover 1st edition 1991

Printed on acid-free paper.

Contents

List of Figures

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 programing 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 computers. 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