A SYSTOLIC ARRAY OPTIMIZING COMPILER

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:

Logic Minimization Algorithms for VLSI Synthesis. R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. ISBN 0-89838-164-9.
Adaptive Filters: Structures, Algorithms, and Applications. M.L. Honig and D.G. Messerschmitt. ISBN 0-89838-163-0.
Introduction to VLSI Silicon Devices: Physics, Technology and Characterization. B. El-Kareh and R.J. Bombard. ISBN 0-89838-210-6.
Latchup in CMOS Technology: The Problem and Its Cure. R.R. Troutman. ISBN 0-89838-215-7.
Digital CMOS Circuit Design. M. Annaratone. ISBN 0-89838-224-6.
The Bounding Approach to VLSI Circuit Simulation. C.A. Zukowski. ISBN 0-89838-176-2.
Multi-Level Simulation for VLSI Design. D.D. Hill and D.R. Coelho. ISBN 0-89838-184-3.
Relaxation Techniques for the Simulation of VLSI Circuits. J. White and A. Sangiovanni-Vincentelli. ISBN 0-89838-186-X.
VLSI CAD Tools and Applications. W. Fichtner and M. Morf, editors. ISBN 0-89838-193-2.
A VLSI Architecture for Concurrent Data Structures. W.J. Dally. ISBN 0-89838-235-1.
Yield Simulation for Integrated Circuits. D.M.H. Walker. ISBN 0-89838-244-0.
VLSI Specification, Verification and Synthesis. G. Birtwistle and P.A. Subrahmanyam. ISBN 0-89838-246-7.
Fundamentals of Computer-Aided Circuit Simulation. W.J. McCalla. ISBN 0-89838-248-3.
Serial Data Computation. S.G. Smith and P.B. Denyer. ISBN 0-89838-253-X.
Phonological Parsing in Speech Recognition. K.W. Church. ISBN 0-89838-250-5.
Simulated Annealing for VLSI Design. D.F. Wong, H.W. Leong, and C.L. Liu. ISBN 0-89838-256-4.
Polycrystalline Silicon for Integrated Circuit Applications. T. Kamins. ISBN 0-89838-259-9.
FET Modeling for Circuit Simulation. D. Divekar. ISBN 0-89838-264-5.
VLSI Placement and Global Routing Using Simulated Annealing. C. Sechen. ISBN 0-89838-281-5.
Adaptive Filters and Equalisers. B. Mulgrew and C.F.N. Cowan. ISBN 0-89838-285-8.
Computer-Aided Design and VLSI Device Development, Second Edition. K.M. Cham, S.-Y Oh, J.L. Moll, K. Lee, P.Vande Voorde and D. Chin. ISBN 0-89838-277-7.
Automatic Speech Recognition. K-F.Lee. ISBN 0-89838-296-3.
Speech Time-Frequency Representations. M.D. Riley. ISBN 0-89838-298-X.
Algorithms and Techniques for VLSI Layout Synthesis. D. Hill, D. Shugard, J. Fishburn and K. Keutzer. ISBN 0-89838-301-3.

A SYSTOLIC ARRAY OPTIMIZING COMPILER

by

Monica S. Lam
Carnegie Mellon University

with a foreword by

H.T. Kung

KLUWER ACADEMIC PUBLISHERS
Boston/Dordrecht/London

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

Distributors for the UK and Ireland:
Kluwer Academic Publishers
Falcon House, Queen Square
Lancaster LA1 1RN, UNITED KINGDOM

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

Library of Congress Cataloging-in-Publication Data

Lam, Monica S.
A systolic array optimizing compiler.

(Kluwer international series in engineering and computer science. VLSI, computer architecture, and digital signal processing)
Includes index.
1. Compilers (Computer programs) 2. Systolic array circuits. I. Title. II. Series.
QA76.76.C65L36 1989 005.4'53 88-13904
ISBN-13: 978-1-4612-8961-6 e-ISBN-13: 978-1-4613-1705-0
DOI: 10.1007/978-1-4613-1705-0

To my sister Angie

Table of Contents

List of Figures

List of Tables

Foreword by H. T. Kung

Generally speaking, computer designers use two methods to build high performance computers. One is to use fast hardware components. But this approach has the drawback that high speed parts may be expensive and sometimes may even have to be custom made. The second method, which avoids this drawback, is to use parallel architectures. That is, the hardware provides multiple functional units that can operate simultaneously. As today's high performance machines are already approaching the speed limit of hardware components, more and more designs will have to rely on parallel architectures for the next level of speed up.

To use a machine based on a parallel architecture, we need to schedule the multiple functional units in the machine to implement desired computations. This scheduling task is one of the well known difficult problems in computer science. Monica Lam in this monograph describes a compiler that shield most of the difficulty from the user, and can generate very efficient code.

These results are based on Monica Lam's implementation of an op-

timizing compiler on the Warp systolic computer at Carnegie Mellon University. She has extended the state-of-the-art compiler techniques such as software pipelining, and devised several new optimizations. These are clearly described and carefully evaluated in the monograph.

I hope that more books of this nature, that are based on real experiences on real computer systems, can be published. These books are especially appreciated in graduate courses in computer systems. I compliment Kluwer's efforts in this area.

H. T. Kung

Preface

This book is a revision of my Ph.D. thesis dissertation submitted to Carnegie Mellon University in 1987. It documents the research and results of the compiler technology developed for the Warp machine.

Warp is a systolic array built out of custom, high-performance processors, each of which can execute up to 10 million floating-point operations per second (10 MFLOPS). Under the direction of H. T. Kung, the Warp machine matured from an academic, experimental prototype to a commercial product of General Electric. The Warp machine demonstrated that the scalable architecture of *high-performance*, *programmable* systolic arrays represents a practical, cost-effective solution to the present and future computation-intensive applications. The success of Warp led to the follow-on iWarp project, a joint project with Intel, to develop a single-chip 20 MFLOPS processor. The availability of the highly integrated iWarp processor will have a significant impact on parallel computing.

One of the major challenges in the development of Warp was to build an optimizing compiler for the machine. First, the processors in the

array cooperate at a fine granularity of parallelism, interaction between processors must be considered in the generation of code for individual processors. Second, the individual processors themselves derive their performance from a VLIW (Very Long Instruction Word) instruction set and a high degree of internal pipelining and parallelism. The compiler contains optimizations pertaining to the array level of parallelism, as well as optimizations for the individual VLIW processors. The compiler has been used extensively within and outside the Carnegie Mellon University research community, and the quality of the code produced has been surprisingly good.

This research confirms that compilers play a major role in the development of novel high-performance architectures. The Warp compiler was developed concurrently with the Warp architecture. It provided essential input to the architecture design that made Warp usable and effective.

Acknowledgments

I want to thank H. T. Kung for the past years of advice; I am fortunate to have him as an advisor. He has taught me much about many different aspects of research, and has created an exciting project and a productive research environment. His guidance and enthusiasm helped make my thesis work both rewarding and enjoyable.

I would also like to express my thanks to Thomas Gross. He has put in tremendous efforts into the compiler, in both the technical and managerial aspects. Together with the rest of the compiler team, C. H. Chang, Robert Cohn, Peter Lieu, Abu Noaman, James Reinders, Peter Steenkiste, P. S. Tseng, and David Yam, he transformed my code generator from an academic experiment into a real-life system.

All members of the Warp project have contributed to this research. Thanks go to the hardware team for making a machine that runs reliably. In particular, Onat Menzilcioglu and Emmanuel Arnould had done a tremendous job designing and building the Warp cell and the interface unit. I wish to thank Marco Annaratone for his dedication in his work on the external host, and Bernd Bruegge for creating a fine programming

environment. I have to thank everybody in the project for using and putting up with the W2 compiler. Jon Webb is one of our primary customers; if not for him, I would not have the 72 programs for evaluating the compiler.

I want to thank Mosur Ravishankar, who has painstakingly read through drafts of my thesis, and has given me plenty of useful comments. I'd like to thank my colleagues and friends for a wonderful time at Carnegie Mellon University. Last, but not least, my thanks go to my parents and family for their support and love.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

1
Introduction

Since Kung and Leiserson introduced "systolic arrays" in 1978, tremendous amounts of research effort have gone into systolic algorithm research. Over the last decade, researchers have developed many systolic algorithms in many important numerical processing areas, including signal processing and scientific computing, demonstrating that systolic architecture is highly suited to deliver the bandwidth required of those applications. The original concept of systolic computing is to map systolic algorithms directly onto custom hardware implementations, and the cost of implementation has limited the practice of systolic computing to a few instances of dedicated systolic hardware. Most systolic algorithms remained as paper designs.

The CMU's Programmable Systolic Chip (PSC) project was the first to address some of the lower level architecture and implementation issues of programmable systolic arrays, and proposed a building block for a set of systolic algorithms. This work showed that there was a wide gap between the theoretical systolic array machine model and practical,

programmable systems and exposed many practical issues that must be addressed before systolic computing could become a reality. For example, floating-point arithmetic capability is absolutely necessary for systolic arrays to be employed in signal processing or scientific computing. The cost of hardware logic to implement floating-point arithmetic makes programmability even more attractive.

The introduction of single-chip floating-point processors at the end of 1983 sparked off the design of the Warp machine here at Carnegie Mellon. The Warp machine is a high-performance, programmable systolic array. With each processor capable of a peak computation rate of 10 million floating-point operations per second (10 MFLOPS), a 10-cell array can deliver a peak aggregate bandwidth of 100 MFLOPS. With the assistance of our industrial partner, General Electric, the first 10-cell array was completed in early 1986. The Warp machine is the first commercial programmable systolic array.

The Warp machine is an important landmark in systolic computing research. It is an execution vehicle for the numerous systolic algorithms that have been designed and never implemented. More importantly, we have greatly extended the concept of systolic computing and showed that systolic arrays of *programmable* and *high-performance* processors can deliver execution speeds that rival existing supercomputers in the numerical processing arena.

To integrate the concepts of programmability and high-performance cells into systolic processing, we carefully scrutinized every characteristic typical of theoretical systolic array models with considerations to hardware, software and applications. As a result, the Warp machine represents a significant departure from the conventional systolic array architecture. The Warp architecture retains the high bandwidth, low latency inter-cell communication characteristic, and thus possesses the scalability property of systolic arrays. On the other hand, many of the assumed systolic array characteristics were constraints imposed by direct

implementation of the algorithm in custom VLSI circuitry. As an example, to keep the logic simple and regular, all cells in the array must repeat the same operation in lock step throughout the computation. Such constraints were removed whenever feasible. This architecture study is significant in fleshing out the abstract machine model and updating the model with realistic implementation constraints.

But what is the cost of programmability? Systolic arrays are known for their efficiency; can a programmable machine be as efficient as dedicated hardware? To ensure that the machine has comparable raw processing capability, the Warp cells employ optimization techniques such as pipelining and parallel functional units, and the multiple functional units are directly accessible to the user through a VLIW (Very Long Instruction Word) instruction set. Each individual functional unit is controlled by a dedicated field in the instruction word. There is little difference between this architecture and a microprogrammed, dedicated machine. From the hardware standpoint, the architecture is programmable and the cost of additional hardware to support programmability is low.

In practice, however, a machine can be called programmable only if applications can be developed for the machine with a reasonable effort. With its wide instruction words and parallel functional units, each individually Warp cell is already difficult to program. Moreover, the multiple cells in the array must be coordinated to cooperately in a fine-grained manner in a systolic algorithm. Can we generate programs that efficiently use the parallelism at both the array and the cell level? This question is the focus of this work.

The thesis of this work is that systolic arrays of high-performance cells can be programmed effectively using a high-level language. The solution has two components: (1) a systolic array machine abstraction for the user and the supporting compiler optimizations and hardware features, and (2) code scheduling techniques for general VLIW processors.

The user sees the systolic array as an array of sequential processors communicating asynchronously. This machine abstraction allows the user to concentrate on high-level systolic algorithm design; the full programmability of the processor is accessible to the user through high-level language constructs. Optimizing compilation techniques translate such user programs into efficient code that fully exploits the machine characteristics of the high-performance processors. These techniques include systolic array specific optimizations as well as general scheduling techniques for highly pipelined and parallel processors.

1.1. Research approach

The ideas presented in this book have been implemented and validated in the context of the Warp project. The project started in 1984. The architecture of the machine has gone through three major revisions: a 2-cell prototype was constructed in June 1985, two identical 10-cell prototypes were completed by our industrial partners, General Electric and Honeywell, in the first quarter of 1986, and the first production machine arrived in April 1987. The production version of the Warp machine is currently being marketed by General Electric. Since 1986, in collaboration with Intel Corporation, we have improved the architecture, and started the design and development of a single-chip processor called iWarp. The component is expected to be available by the fourth quarter of 1989.

The research presented here consists of several components. First, we have developed a new machine abstraction for systolic arrays, and captured the abstraction by a programming language called W2. This language hides the internal parallelism and pipelining of the cells. The abstraction was designed together with the code scheduling techniques to ensure that effective code can be generated. The scheduling algorithms were implemented as part of a highly optimizing compiler supporting the language. In the course of the development of the machine abstraction and compiler, necessary hardware support has been identified, some of

which was incorporated in the prototypes and some in the production machine. Finally, a large set of performance data on the machine and the compiler has been gathered and analyzed.

The compiler has been functional since the beginning of 1986. It has been used extensively in many applications such as low-level vision for robot vehicle navigation, image and signal processing, and scientific computing [3, 4]. An average computation rate of 28 MFLOPS is achieved for a sample of 72 application programs. They are all programs with unidirectional data flow that achieve a linear speed up with respect to the number of cells used. Analysis of the programs shows that, on average, the scheduling techniques speed up the programs by about a factor of three.

1.2. Overview of results

The results of this research have two major components: a machine abstraction for systolic arrays, and compiler code scheduling techniques. While this research is undertaken in the context of the Warp machine, the results are applicable to other architectures. The proposed machine abstraction and array level optimizations are useful for systolic array synthesis where either the algorithm and/or machine model is too complex for existing automatic synthesis techniques. The cell level optimizations are applicable for high-performance processors whose internal pipelining and parallelism are accessible at the machine instruction level.

1.2.1. A machine abstraction for systolic arrays

This work proposes an intermediate machine abstraction in which computation on each cell is made explicit and cells communicate asynchronously. That is, a cell communicates with its neighbors by receiving or sending data to a dedicated queue; a cell is blocked when trying to send to a full queue or receive from an empty queue. The

abstraction that each cell is a simple sequential processor allows the user, or a higher level tool, to concentrate on the utilization of the array. Its asynchronous communication primitives allow the user to express the interaction between cells easily.

A surprising result is that while the asynchronous communication model provides a higher level of abstraction to the user than synchronous models, it is also more amenable to compiler optimizations. The high-level semantics of the asynchronous communication model is exploited by the compiler to relax the sequencing constraints between communication operations within a cell. Representing only those constraints that must be satisfied in a manner similar to data dependency constraints within a computation, general code motions on the communication operations can be applied to minimize the execution time on each cell. This approach allows us to generate highly efficient code for systolic arrays of unidirectional data flow from complex systolic programs.

The asynchronous communication model is useful, and in fact recommended, even for hardware implementations without direct support for dynamic flow control. Many systolic array algorithms, including all previously published ones, can be implemented without this dynamic flow control. We can first optimize the program by pretending that there is dynamic flow control; compile-time flow control is then provided using an efficient algorithm.

1.2.2. Cell level optimizations

The high computation rate of the Warp processor is derived from its heavily pipelined and parallel data path and a horizontal instruction format. The data path consists of multiple pipelined functional units, each of which is directly controlled by a dedicated field in the machine instruction. Computation that can proceed in parallel in a single cycle includes seven floating-point multiplications, seven floating-point ad-

ditions, eight register accesses, two memory accesses, four data queue accesses and one conditional branch operation. The potential processing capacity available in this machine organization is tremendous. The code scheduling problem for the Warp processors has been studied in the context of VLIW (very long instruction word) architectures [20], and trace scheduling has been the recommended approach [19]. This work suggests an alternative approach to scheduling VLIW processors: *software pipelining* and *hierarchical reduction.*

Software pipelining is a scheduling technique that exploits the repetitive nature of innermost loops to generate highly efficient code for processors with parallel, pipelined functional units [47, 56]. In software pipelining, an iteration of a loop in the source program is initiated before its preceding iteration is completed; thus at any time, multiple iterations are in progress simultaneously, in different stages of the computation. The steady state of this pipeline constitutes the loop body of the object code.

The drawback of software pipelining is that the problem of finding an optimal schedule is NP-complete. There have been two approaches to software pipelining: (1) change the architecture, and thus the characteristics of the constraints, so that the problem is no longer NP-complete, (2) use software heuristics. The first approach is used in the polycyclic architecture; a specialized crossbar is proposed to make optimizing loops without data dependencies between iterations tractable [47]. However, this hardware feature is expensive to build; and, when inter-iteration dependency is present within the loop, exhaustive search on the strongly connected components of the data flow graph is still necessary [31]. The second approach is used in the FPS-164 compiler [56]. However, software pipelining is applied to a limited class of loops, namely loops that contain only a single Fortran statement.

This research shows that software pipelining is a practical, efficient, and general technique for scheduling the parallelism in a VLIW machine.

We have extended previous results of software pipelining in two ways. First, we show that near-optimal results can be obtained for all loops using software heuristics. We have improved and extended previous scheduling heuristics and introduced a new optimization called *modulo variable expansion*. The latter has part of the functionality of the specialized hardware proposed in the polycyclic machine, thus allowing us to achieve similar performance.

Second, software pipelining has previously been applied only to loops with straight-line loop bodies. In this work, we propose a *hierarchical reduction* scheme whereby entire control constructs are reduced to an object similar to an operation in a basic block. With this scheme, software pipelining can be applied to arbitrarily complex loops. The significance is threefold: All innermost loops, including those containing conditional statements, can be software pipelined. If the number of iterations in the innermost loop is small, we can software pipeline the second level loop as well to obtain the full benefits of this technique. Lastly, hierarchical reduction diminishes the penalty of start-up cost of short vectors.

Analysis of a large set of user programs shows that the combination of software pipelining and hierarchical reduction is effective. Optimal performance is often obtained for many simple, classical systolic algorithms. Many new and more complex systolic programs have been developed using the compiler, and the quality of the generated code is comparable to that of hand crafted microcode.

1.3. This presentation

To establish the context for this book, we first describe the architecture of the Warp systolic array. Chapter 2 describes the design of the architecture and the rationale behind the design decisions. It also highlights the implications of the machine features on the complexity in programming the machine.

Chapter 3 presents the machine abstraction proposed for programmable systolic arrays. In this chapter, we first argue that the user must have control over the mapping of a computation onto the array if efficient code is to be obtained for the complete computation domain of a programmable, high-performance systolic array. Next, we explain the interaction between the internal timing of a cell and the efficiency of a systolic algorithm. We then compare several existing communication models and show that the asynchronous communication model is superior for both reasons of programmability and amenability to optimization. We also discuss the hardware and software support for this communication model.

The fourth chapter introduces the W2 programming language, and presents an overview of the compiler. It describes the different modules of the compiler and their interactions. The main purpose of this chapter is to prepare the ground for discussion of the cell level optimizations in the next two chapters: Chapter 5 describes software pipelining and Chapter 6 describes hierarchical reduction.

In Chapter 5, we concentrate on the technique of software pipelining, and describe the algorithm for scheduling innermost loops whose bodies consist only of a single basic block. We present the mathematical formulation of the problem, describe our scheduling heuristics, introduce a new optimization called modulo variable expansion, and compare our approach to existing hardware and software approaches to the problem.

Chapter 6 presents the hierarchical reduction technique; this technique allows us to model entire control constructs as simple operations in a basic block. This chapter shows how this approach makes list scheduling applicable across basic blocks, and software pipelining applicable to all loops. In this chapter, we also compare our overall scheduling strategy (using both software pipelining and hierarchical reduction) with other methods proposed for handling high degrees of parallelism and pipelining in the data path: trace scheduling [19] and vector processing.

Chapter 7 presents the performance data on the Warp machine and the compiler. We first present experimental data on a collection of 72 user programs. There are two parts to the evaluation: measurements on entire programs and measurements on the innermost loops of the programs. In both parts of the experiment, we compare the performance of the code against unoptimized code to study the significance of the scheduling techniques, and we also compare the performance of the code with a theoretical upper bound to measure the efficiency of the code. Second, we present the Livermore Loop benchmark results on single Warp cells. Besides analyzing the efficiency of the compiler, we also present some insights on the effectiveness of the architecture.

Lastly, Chapter 8 summarizes the results of the research and presents the conclusions of this work. We present the conclusion that the concept of systolic processing can be fruitfully applied to an array of high-performance processors.

2
Architecture of Warp

The Warp project is a study in *high-performance, programmable systolic* architectures. Although the feasibility of the concept of systolic computing has long been established, the results have mainly been theoretical in nature, and many lower-level architectural and implementation issues have not been studied and addressed. We believed that these practical issues would not even be exposed, let alone resolved, without implementation and experimentation. We did not just set out to implement a systolic array to perform a specific function, we extended the concept of systolic processing by implementing each processing unit as a programmable, high-performance processor. The tenet of the project is that the theories and results in systolic computing research are applicable also to arrays of processors that are programmable and powerful.

The Warp array represents a serious departure from the traditional systolic machine model; it is a result of a careful analysis and re-evaluation of every feature that has always been associated with systolic processing. Each processor is implemented as an individually

programmable micro-engine, with a large local data memory. It achieves a high peak computational bandwidth by allowing direct access of its multiple functional units through a wide instruction word. It has a high communication bandwidth typical of systolic arrays; but more importantly it contains architectural features that allow the bandwidth to be used effectively. The Warp array does not only serve as a vehicle for implementing existing systolic algorithms, its flexibility and programmability also open up possibilities for new parallel algorithms.

The architecture of Warp has gone through several revisions: two prototype versions, both implemented on wire-wrapped boards, and the production machine implemented on printed circuit boards. In the following, we refer to the production machine as the PC machine, for short. These two architectures are followed on by iWarp, a single-chip processor with a much improved architecture. The goal of the prototype machines was to demonstrate the feasibility of such a system, and its target application domain was quite limited. The experience gained in using the prototypes enabled us to implement a production machine that can support a larger computation domain. The architecture of the integrated version is a complete redesign to take advantage of the new implementation technology. The availability of the iWarp processor would greatly reduce the size and cost, and enhance the reliability, of complete systems.

The compiler research has been instrumental in the development of the architecture of Warp. Preliminary studies of the compiler design have contributed to the design of the prototypes. Insights gained from the construction of the compiler as well as the evaluation of the compiler generated code were incorporated into the PC machine. This chapter presents the architecture of the 10-cell prototype system in detail, as this is the architecture on which the research is based. Major revisions to the prototype in designing the PC machine are also described. This chapter also discusses the application domain of the Warp architecture, as well as its programming complexity.

2.1. The architecture

Warp is integrated into a general purpose host as an attached processor. There are three major components in the system—the Warp processor array (*Warp array*), the interface unit (*IU*), and the *host*, as depicted in Figure 2-1. The Warp array performs computation-intensive routines such as low-level vision routines or matrix operations. The IU transfers data between the array and the host; it also generates addresses for local cell memories and systolic control signals used in the computation of the cells. The host supplies data to and receives results from the IU, in addition to executing the parts of the application programs that are not mapped onto the Warp array. For example, the host performs decision-making processes in robot navigation, beam adaptation in sonar processing, and evaluation of convergence criteria in iterative methods for solving systems of linear equations.

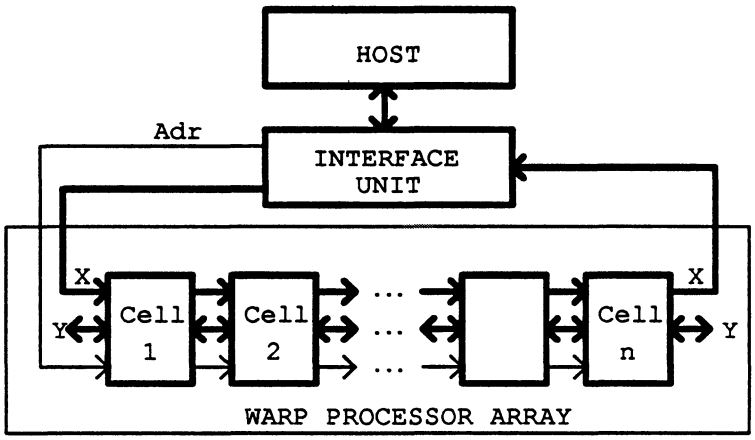


Figure 2-1: Warp system overview

The Warp array is a one-dimensional systolic array with identical cells called Warp cells. Data flow through the array on two data paths (X and Y), while addresses (for local cell memories) and systolic control signals travel on the Adr path (as shown in Figure 2-1). Each cell can transfer up to 20 million 32-bit words (80 Mbytes) per second to and

from its neighboring cells, in addition to propagating 10 million 16-bit addresses across the array per second. The Y data path is bidirectional, with the direction configurable statically.

2.1.1. Warp cell

Each Warp cell is a high-performance processor, with its own sequencer and program memory. It has a wide machine instruction format: units of the data path are directly controlled through dedicated fields in the instruction. The data path of a processor in the prototype array is illustrated in Figure 2-2. It consists of a 32-bit floating-point multiplier

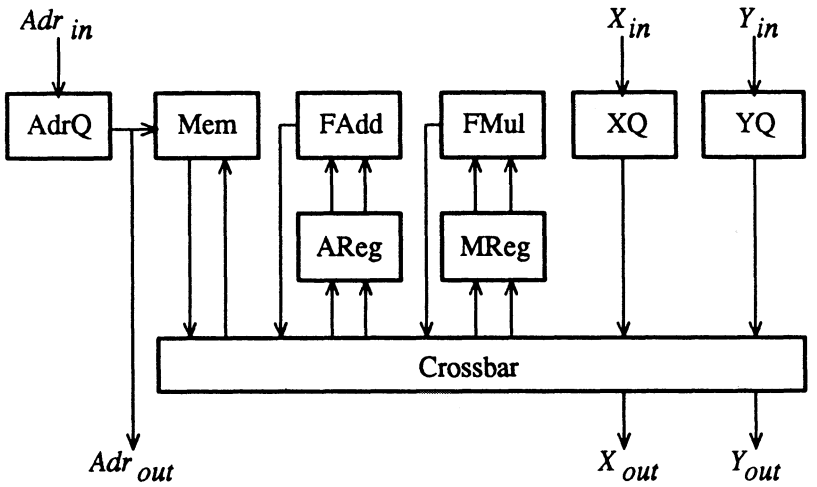


Figure 2-2: Data path of a Warp cell

(Mpy) and a 32-bit floating-point adder (Add), a local memory for resident and temporary data (Mem), a queue for each inter-cell communication path (X, Y, and Adr Queue), and a register file to buffer data for each floating-point unit (AReg and MReg). All these components are interconnected through a crossbar switch. The instructions are executed at the rate of one instruction every major clock cycle of 200 ns. The details of each component of the cell, and the differences between the prototype and the production machine, are given below.

Floating-point units. The floating-point multiplier and adder are implemented with commercial floating-point chips [59]. At the time of design for the Warp machine, these floating-point parts depended on extensive pipelining to achieve high performance. Both the adder and multiplier have 5-stage pipelines.

Data storages. The cell memory hierarchy includes a data memory and two 4-ported register files, one for each functional unit. The size of the data memory of the prototype is 4 Kwords, and that of the PC machine is 32 Kwords. In an instruction, two memory accesses can be made, one read and one write. The register files serve as buffers for the operands of the multiplier and the adder. The register files hold 31 words of data each. (Actually the register files hold 32 words; it is written to in every clock, so one word is used as a sink for those instructions without useful write operations). An instruction can read two words from and write two words into each register file. There is a two-clock delay associated with the register file. Therefore, the result of a floating-point operation cannot be used until a total of seven clock ticks later.

Address generation. The Warp cells in the prototype machine are not equipped with integer arithmetic capability. The justification is that systolic cells typically perform identical, data-independent functions, using identical addressing patterns and loop controls. For example, when multiplying two matrices, each cell is responsible for computing several columns of the results. All cells access the same local memory location, which has been loaded with different columns of one of the argument matrices. Therefore, common addresses and loop control signals can be generated externally in the IU and propagated to all the cells. Moreover, it is desirable that each Warp cell performs two memory references per instruction. To sustain this high local memory bandwidth, the cell demands a powerful address generation capability, which was expensive to provide. We could dedicate much more hardware resources to address generation by implementing it only once in the IU, and not replicated on all Warp cells. Although all cells must execute the same program and

use the same addresses, they do not necessarily operate in lock step. As we shall show, cells in a systolic array often operate with a time delay between consecutive cells, The queue along the address path allows them to operate with different delays at different times.

The lack of address generation capability on the cells means that cells cannot execute different programs efficiently. The finite queue size in the address path also places a limitation on programs in which cells execute the same code, since the delay between cells is limited by the length of the queue. At the time the PC was designed, a new address generation part became available. This part can supply two addresses every clock and includes a 64-register file. As the cost of address generation capability is significantly lowered, it is provided on each cell in the PC Warp.

Inter-cell communication. Each cell communicates with its left and right neighbors through point-to-point links, two for data and one for addresses and control signals. A queue is associated with each link and is placed on the input side of each cell. This queue is 128 and 512 words deep in the prototype and the PC Warp machines, respectively. The queues are important because they provide a buffer for data communicated between cells, and thus allow the communicating cells to have different instantaneous input/output rate.

In the prototype machines, flow control is not provided in hardware. That is, the software must ensure that a cell never reads from an empty queue or writes into a full queue. Dynamic flow control is provided in the PC Warp: a cell is blocked whenever it tries to read from an empty queue or write to a full queue.

Crossbar switch. Experience with the Programmable Systolic Chip showed that the internal data bandwidth is often the bottleneck of a systolic cell [22]. In the Warp cell, the two floating-point arithmetic units can consume up to four data items and generate two results per instruc-

tion. The data storage blocks are interconnected with a crossbar to support this high data processing rate. There are six input and eight output ports connected to the crossbar switch; six data items can be transferred in a single clock, and an output port can receive any of the data items.

Control path. Each Warp cell has its own local program memory and sequencer. Even if the cells execute the same program, it is not easy to broadcast the microinstruction words to all the cells, or to propagate them from cell to cell, since the instructions are very wide. Moreover, although the cell programs may be the same, cells often do not execute them in lock step. The local sequencer also supports conditional branching efficiently. In SIMD machines, branching is achieved by masking. The execution time is equivalent to the *sum* of the execution time of both branches. With local program control, different cells may follow different branches of a conditional statement depending on their individual data; the execution time is the *maximum* of the execution time of both branches.

The data path is controlled by wide instruction words, with each component controlled by a dedicated field. In a single instruction, a processor in the Warp prototype array can initiate two floating-point operations, read and write one word of data from and to memory, read four words and write four words to the register files, receive and send two data items from and to its two neighbors, and conditionally branch to a program location. In addition, the PC machine can perform two integer operations, which include reading the operands and storing the results into the register file of the address generation unit. The orthogonal instruction organization makes scheduling easier since there is no interference in the schedule of different components.

2.1.2. Interface unit

The IU generates data memory addresses and loop control signals for the cells. In each instruction, the IU can compute up to two addresses, modify two registers, test for the end of a loop, and update a loop counter. To support complex addressing schemes for important algorithms such as FFT, the IU also contains a table of pre-stored addresses; this table can be initialized when the microcode is loaded. The IU has the ability to receive four 8-bit or two 16-bit data packed into one 32-bit word. Each data is unpacked and converted to a 32-bit floating-point data before it is sent to the Warp array. The opposite transformation is available as well.

2.1.3. The host system

The Warp host, depicted in Figure 2-3, consists of a SUN-3 workstation (the master processor) running UNIX and an external host. The workstation provides a UNIX environment to the user. The external host consists of two *cluster processors* and a *support processor*. The processors in the external host run in stand-alone mode. The support processor controls peripheral I/O devices (such as graphics boards), and handles floating-point exceptions and other interrupt signals from the Warp array. The two clusters buffer the data and results to and from the array. They work in tandem, each handling a unidirectional flow of data to or from the Warp processor, through the IU. The two clusters can exchange their roles in sending or receiving data in different phases of a computation, in a ping-pong fashion.

The external host is built around a VME bus. The two clusters and the support processor each consists of a stand-alone MC68020 microprocessor (P) and a dual-ported memory (M), which can be accessed either via a local bus or via the global VME bus. The local bus is a VSB bus in the PC Warp machine and a VMX32 bus for the prototype;

the major improvements of VSB over VMX32 are better support for arbitration and the addition of DMA-type accesses. Each cluster has a *switch* board (S) for sending and receiving data to and from the Warp array, through the IU. The switch also has a VME interface, used by the master processor to start, stop, and control the Warp array. The VME bus of the master processor inside the workstation is connected to the VME bus of the external host via a bus-coupler. The total memory in the external host can be up to 36 Mbytes.

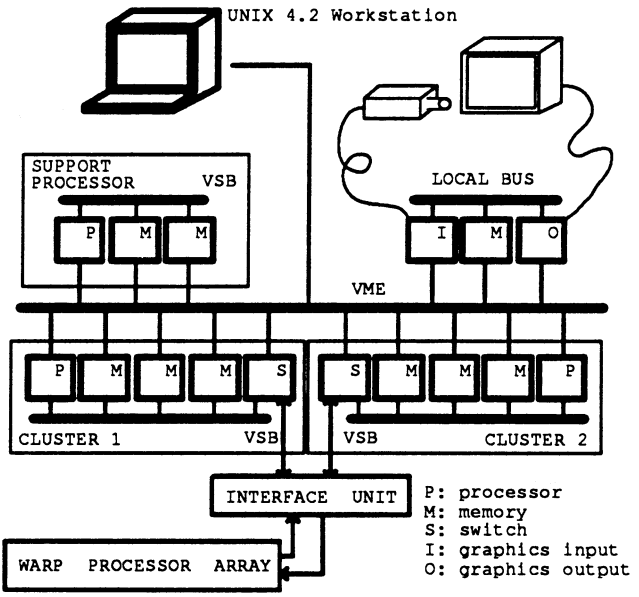


Figure 2-3: Host of the Warp machine

2.2. Application domain of Warp

The architecture of the Warp array can support various kinds of algorithms: fine-grain or coarse-grain parallelism, local or global operations, homogeneous or heterogeneous computing. (An algorithm is homogeneous if all cells execute the same program, and heterogeneous otherwise.) The factors contributing to the versatility of Warp include: simple topology of a linear array, powerful cells with local program con-

trol, large data memory and high inter-cell communication bandwidth. These features support several important problem decomposition methods [30, 37, 38].

The configuration of a linear array is easy to implement in hardware, and demands the smallest external I/O bandwidth, since only the two end-cells communicate with the outside world. Also, it is easier to use than other higher-dimensional arrays. Many algorithms have been developed for linear arrays in the scientific computing domain. Our experience of using Warp for low-level vision has also shown that a linear organization is suitable in the vision domain as well. Moreover, other interconnection topologies can be efficiently simulated on a linear array, since each Warp cell is a powerful, programmable processor. For example, a single Warp cell can be time multiplexed to perform the function of a column of cells, so that the linear array can simulate a two-dimensional systolic array.

Warp can be used for both fine-grain and coarse-grain parallelism. It is efficient for fine-grain parallelism typical of systolic processing, because of its high inter-cell bandwidth. The I/O bandwidth of each cell is higher than other processors of similar computation power, and it supports the transfer of large volumes of intermediate data between neighboring cells. Warp is also efficient for coarse-grain parallelism because of its powerful cells. With its own program sequencer, program memory and data memory, each cell is capable of operating independently. The data memory sizes (4 Kwords in the prototype machine, and 32 Kwords in the PC machine) are large for systolic array designs. It has been shown that the I/O requirement to sustain a high computation rate is lower for machines with larger data memories [35].

Systolic arrays are known to be effective for *local* operations, in which each output depends only on a small corresponding area of the input. Warp's large memory and high I/O bandwidth enable it to perform *global* operations in which each output depends on any or a large

portion of the input [37]. Examples of global operations are FFT, component labeling, Hough transform, image warping, and matrix computations such as matrix multiplication or singular value decomposition (SVD). The ability to perform global operations as well significantly extends the computation domain of the machine.

2.3. Programming complexity

While the potential performance of a high-performance systolic array like Warp is enormous, the complexity of using the machine is proportionally overwhelming. Parallelism exists at several levels in the Warp architecture. At the system level, there are separate processors for input/output (the external host), control (IU), and data computation (Warp array). At the array level, there are ten cells tightly coupled together to solve a single problem; and finally at the cell level, there is a horizontal architecture with multiple pipelined functional units.

System level. Communication between the host and the processor array requires code to be generated for all three components of the system. Data must first be transferred from the host to the memory of the input cluster processor. The input processor then sends the data to the IU in the order the data is used in the Warp cells. The IU and the cluster memories communicate asynchronously via a standard bus. The IU feeds the data received from the cluster to the first Warp cell. The output process, however, requires cooperation in software only between the cell outputting the data and the host. The outputting cell tags the data with control signals for the IU output buffer, and the host reads the data directly from the buffer via the standard bus. In the prototype machines, where there is no flow control provided in the queues on the cells, the IU must send the data to the first cell before it tries to remove data from the queue. If the cluster processor cannot supply data in time, the entire Warp array is stalled.

For the prototype machine, since the cells do not have local address

generation capability, they must cooperate closely with the IU. Data independent addresses and loop control signals used in the cells are computed on the IU and propagated down to the individual cells. Since addresses and loop controls are integral to any computation, the actions taken by the IU and the Warp cells are strongly coupled.

Massive amounts of detail must be mastered to use the Warp system. Fortunately, the problem of using this level of parallelism is not intrinsically hard, and can be solved through careful bookkeeping. This problem will be addressed in Chapter 4 where the overall structure of the compiler is described.

Array level. The mapping of a computation onto a locally connected array of processors has been an active area of research for the last several years. Systolic array designs require the workload be partitioned evenly across the array. Any shared data between cells must be explicitly routed through the limited interconnection in the array. The computation and the communication must be scheduled such that the cells are utilized as much as possible. Automatic synthesis techniques have been proposed only for simple application domains and simple machine models. The problem of using the array level concurrency of a high-performance array effectively is an open issue.

The approach adopted in this work is to expose this level of concurrency to the users. The user specifies the high-level problem decomposition method and the compiler handles the low-level synchronization of the cells. The justification of the approach and the exact computation model are presented in the next chapter.

Cell level. The 10 MFLOPS peak computation rate of a Warp processor is achieved through massive pipelining and parallelism. Sophisticated scheduling techniques must be employed to use this level of concurrency effectively. For example, a multiply-and-accumulate step takes 14 instructions on the Warp cell; the full potential of the cell can be

achieved only if 14 such steps are overlapped. This scheduling problem has been studied in the context of horizontal microcode compaction [23], array processors [44, 47, 56], and VLIW (Very Long Instruction Word) machines [20]. Details on this level of parallelism and the proposed scheduling techniques are described in Chapters 5 and 6.

3
A Machine Abstraction

This chapter studies the design of a machine abstraction for systolic arrays with high-performance cells. While the study is based on the Warp architecture, the design is applicable to any programmable array for which a high-level language is used to specify fine-grained parallel computation on the cells. In this chapter, we also discuss the hardware and software necessary to support the proposed machine abstraction.

The machine abstraction defines the lowest level details that are exposed to any user of the machine. Therefore, it must be both general and efficient. This chapter first argues that the user must have control over the mapping of a computation across the array to achieve both generality and efficiency. The user should specify the actions for each cell; the language in which the programs are specified, however, should be supportive with high-level constructs.

Next, we compare various communication models for expressing the interaction between cells. High-performance cells typically employ op-

timization techniques such as internal pipelining and parallelism. We show that if we want to achieve efficiency while hiding the complexity of the cell architecture to the user, we must also abstract away the low-level cell synchronization. We, therefore, propose to use an asynchronous communication model, not only because it is easier to use but also because it is more amenable to compiler optimizations.

Lastly, this chapter discusses the hardware and software support for the asynchronous communication model. The asynchronous communication model does not necessarily have to be supported directly in hardware. More precisely, it is not necessary to implement dynamic flow control, the capability to stall a cell when it tries to read from an empty queue, or send to a full queue. Many systolic algorithms, including all previously published ones, can be implemented on machines without dynamic flow control hardware. This chapter describes how compile-time flow control can be provided.

3.1. Previous systolic array synthesis techniques

Systolic algorithm design is nontrivial. The high computation throughput of systolic arrays is derived from the fine-grain cooperation between cells, where computation and the flow of data through the array are tightly coupled. Processors in a systolic array do not share any common memory; any common data must be routed from cell to cell through the limited interconnection in the array. This complexity in designing a systolic algorithm has motivated a lot of research in systolic array synthesis [7, 13, 26, 36, 40, 41, 46].

Automatic synthesis has been demonstrated to be possible for simple problem domains and simple machine models [6, 13, 26, 46]. The target machine model of previous systolic work was custom hardware implementation using VLSI technology. The main concerns were in mapping specific algorithms onto a regular layout of simple, identical hardware components. The computation performed by each cell must

therefore be regular, repetitive and data-independent. The issues that must be addressed for generating code for a linear high-performance systolic array are significantly different. Computation amenable to automatic synthesis techniques can be mapped onto a linear array fairly easily. The issues of using an array of powerful cells are in mapping complex, irregular computations onto the linear configuration, and in allowing for the complex cell timing of a high-performance processor.

The machine model assumed by these array synthesis tools is that each cell performs one simple operation repeatedly. The operation itself is treated as a black box; the only visible characteristics are that in each clock cycle, a cell would input data, compute the result, then output the result at the end of the clock. The result is available as input to other cells in the next clock.

Computations suitable for direct mapping onto silicon must be repetitive and regular. They consist of large numbers of identical units of operations so that the array can be implemented through replication of a basic cell. The data dependencies between these units must also be regular so that the interconnection of these cells do not consume much silicon area and can be replicated easily. Examples of such computations are uniform recurrence equations [46].

The emphasis of the research was in the partitioning of computation onto a two-dimensional array. Although the notation and the techniques used in the different approaches may be different, most of them have a similar basic model: The computation is modeled as a lattice with nodes representing operations and edges representing data dependencies. Each operation node executes in unit time. The lattice is mapped onto a space-time domain; the time and space coordinates of each node indicate when and where in the array to perform the operation. This modeling of the synthesis problem is powerful for mapping a regular computation onto a regular layout of simple cells. On the other hand, these built-in assumptions of regular computation and simple cells make these synthesis tech-

niques unsuitable for high-performance arrays such as the Warp machine.

While computation partitioning often requires insights into the application area, scheduling of cell computation and communication involves mostly tedious bookkeeping. There have been a couple of systolic design tools whose goals are to alleviate the user of this second step. The user is responsible for partitioning problems across the array and expressing algorithms in terms of a higher-level machine abstraction. These tools then convert such algorithms to programs for the low-level hardware. The user can write inefficient algorithms using a simple machine model, and the tools transform them into efficient ones for the complicated hardware.

For example, Leiserson and Saxe's *retiming lemma* gives the user the illusion that broadcasting is possible [41]. It converts all broadcasting signals to local propagation of signals. The *cut theorem* introduced by Kung and myself transforms systolic designs containing cells executing operations in single clock cycles to arrays with pipelined processors [36]. This technique will be described in more details in Section 3.2.2. A limitation common to both tools is that they are designed for algorithms whose communication pattern is constant across time.

A programmable array of powerful cells is capable of a far more general class of problems than those previously studied for systolic arrays. Cells do not have to compute the same function; the function can consist of different operations requiring different amounts of time; the communication pattern does not have to be constant across time. Mapping the more complex problems onto Warp requires techniques that exploit the semantics of the operations in a computation, which were treated as black boxes in previous approaches. Examples include the polynomial GCD computation [5] and connected component labeling [38].

As efficiency and generality are the goals of the machine abstraction, the model must allow the user to control the mapping of computation onto the processors. Higher level transformation tools that can map specific computational models efficiently can be implemented above the machine abstraction.

3.2. Comparisons of machine abstractions

While the decomposition of problem across cells is best handled by the user, the pipelining and parallelism in the data path of the cells can best be utilized by automatic scheduling techniques. (This claim is substantiated in Chapters 5 and 6.) Thus, we propose that a high-level language, complete with typical control constructs, be used to describe the cell computation. We are thus faced with the following question: given that the computation of a cell is to be described in a high-level language oblivious of the low-level timing of the cell, how should the interaction between cells be specified?

There have been several different models proposed for describing point-to-point communication between processes. (We assume that there is precisely one process per processor.) Some models require that the user specify precisely the operations that are to be performed concurrently; examples include the SIMD model (single instruction, multiple data) and the primitive systolic model, explained below.

SIMD is a well established model of computation for processor arrays with centralized sequencing control. In the SIMD model of computation, all cells in an array execute the same instruction in lock step. Communication is accomplished by shift operations; every cell outputs a data item to a neighbor and inputs an item from the neighbor on the opposite side.

The primitive systolic model is commonly used in systolic design descriptions and synthesis techniques. This model has a simple

synchronous protocol. With each beat of the clock, every cell receives data from its neighbors, computes with the data, and outputs the results, which are available as input to its neighbors in the next clock. The same computation is repeatedly executed on every cell in the array. Computation in this model can be easily expressed as an SIMD program where the input, compute and output phases are iterated in a loop. However, this rigid control flow structure can be exploited in optimizing the program to allow for complex internal cell timing, as described below.

Both the SIMD and the primitive systolic models have the potential to be adopted as the communication model for Warp because the Warp architecture shares many similarities with machines using such models. Many programs implemented on the Warp prototype can be written as an SIMD program. We show below that these synchronous models are inappropriate because they are awkward for describing some computation modes that are well supported by the machine and, perhaps surprisingly, they are not amenable to compilation techniques used for high-performance cells.

3.2.1. Programmability

The difficulty in expressing a computation should only reflect the inherent mismatch between the architecture and the computation, and should not be an artifact of the machine abstraction. We have identified two useful problem partitioning methods, the *parallel* and *pipeline* schemes, that are well supported by the Warp architecture [4, 38]. The machine abstraction must support easy expression of algorithms employing these methods. Let us first describe these partitioning methods before we discuss the expressibility of the communication models for programs using these schemes.

3.2.1.1. Partitioning methods

In the parallel mode, each processor computes a subset of the results. In many cases, each processor needs only a subset of the data to produce its subset of results; sometimes it needs to see the entire data set. The former is called *input partitioning*, and the latter is called *output partitioning* [3].

In the input partitioning model, each Warp cell stores a portion of the input data and computes with it to produce a corresponding portion of the output data. Input partitioning is a simple and powerful method for accessing parallelism – most parallel machines support it in one form or another. Many of the algorithms on Warp, including most of the low-level vision routines, make use of it. In the output partitioning model, each Warp cell processes the entire input data set, but produces only part of the output. This model is used when the input to output mapping is not regular or when any input can influence any output. This model usually requires a lot of memory because either the required input data set must be stored and then processed later, or the output must be stored in memory while the input is processed, and then output later. An example of an algorithm that uses output partitioning is Hough transform [3].

In the pipelined mode, the computation is partitioned among cells, and each cell does one stage of the processing. This is the computation model normally thought of as "systolic processing." It is Warp's high inter-cell bandwidth and effectiveness in handling fine-grain parallelism that makes it possible to use this model. An example of the use of pipelining is the solution of elliptic partial differential equations using successive over-relaxation [61]. In the Warp implementation, each cell is responsible for one relaxation [3]. In raster order, each cell receives inputs from the preceding cell, performs its relaxation step, and outputs the results to the next cell. While a cell is performing the k#th relaxation step on row i, the preceding and next cells perform the k–1st and k+1st

relaxation steps on rows i+2 and i–2, respectively. Thus, in one pass through the 10-cell Warp array, the above recurrence is applied ten times. This process is repeated, under control of the external host, until convergence is achieved.

3.2.1.2. Programmability of synchronous models

Consider the following example. Suppose we want to evaluate the polynomial

$$P(x)=C_m x^m+C_{m-1}x^{m-1}+\ldots+C_0$$

for $x_1,\ldots,x_n$. By Horner's rule, the polynomial can be reformulated from a sum of powers into an alternating sequence of multiplications and additions:

$$P(x)=((C_m x+C_{m-1})x+\ldots+C_1)x+C_0$$

The computation can be partitioned using either the parallel or pipelined model. To evaluate the polynomials according to the parallel model, each cell in the array computes $P(x)$ for different values of x. Under the pipeline model, the computation can be partitioned among the cells by allocating different terms in Horner's rule to each cell.

The description of a systolic array for solving polynomials of degree 9 in pipelined mode, using the primitive systolic array model, is given in Figure 3-1. The steady state of the computation is straightforward: In

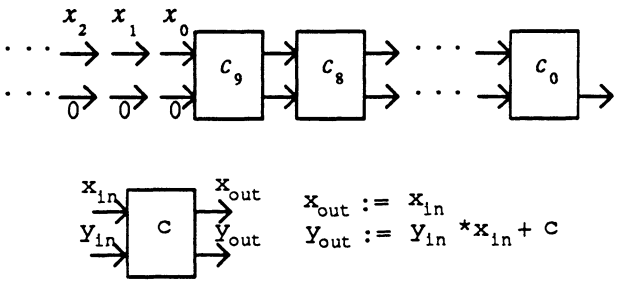


Figure 3-1: Systolic array for polynomial evaluation in pipeline mode

each clock cycle, each cell receives a pair of data, performs a multiplication and an addition, and outputs the results to the next cell. However, the boundary conditions are more complex: only the first cell is supplied with valid data in the first clock cycle, the rest of the cells must wait or compute with invalid data until the first valid one arrives. Since the first result does not emerge from the last cell until the end of the m+1st clock, the computation must be iterated n+m times to calculate the polynomial for n sets of data, and m sets of fictitious data must be tagged on at the end of the input data.

We encounter the same problem when expressing the program in the SIMD model of computation. Using a Pascal-like notation, the program for polynomial evaluation in pipeline mode would look like the following:

```
/* shift in the coefficients */
c := 0.0;
For i := 0 to m do begin
   c := shift (R, X, c);
end;

/*  compute the polynomials  */
xdata := 0.0;
yout := 0.0;
For i := 0 to n+m do begin
   yin := shift (R, Y, yout);
   xdata := shift (R, X, xdata);
   yout := yin * xdata + c;
end;
```

The **shift** operation takes three arguments: direction, channel used, and the value to be shifted out. The direction **R**, or **L**, specifies that data is shifted from the right to the left, or left to right, respectively. The channel, **X** or **Y**, specifies the hardware communication link to be used. Lastly, the value of the third argument is the value sent to the next cell, whereas the result shifted into the cell is returned as the result of the construct. The first loop shifts in the coefficients: the first input is the coefficient for the last cell. The second loop evaluates the polynomials.

In each iteration of the second loop, results from the previous iteration are shifted out and new data for the current iteration are shifted in. Again, to compute the polynomials for n different values of x, the loop has to be iterated $n+m$ times.

The notation is equally inelegant for describing programs that use the parallel mode of decomposition if cells share common data. If the SIMD model of computation is to be used for systolic arrays, then common data must be propagated from cell to cell in the array. The program for evaluating polynomials under the parallel model is:

```
/* shift in the values of x  */
data := 0.0;
For i := 0 to n-1 do begin
   data := shift (R, X, data);
end;

/*  compute the polynomials  */
result := 0.0;
c := 0.0;
For i := 0 to n+m-1 do begin
   c := shift (R, X, c);
   result := result * data + c;
end;

/* shift out the polynomials */
For i := 0 to n-1 do begin
   result := shift (R, X, result);
end;
```

Again, the boundary conditions are complex because not all cells can start computing with meaningful data at the same time. The user must supply fictitious data to the cells and specify which of the outputs of the array constitutes the desired results.

3.2.2. Efficiency

The machine abstraction must permit automatic translation of user's input to efficient code that exploits the potential of the parallel and pipelined resources in each and every cell. It might appear that if the user synchronized the cells explicitly, less hardware or software support would be needed to generate efficient code although the programming task would be more complicated. This is unfortunately not true. Synchronization is related to the internal timing of the cells. If the user does not know the complex timing in the cells, the synchronization performed by the user is not useful. In fact, a primitive communication model not only complicates the user's task, it may also complicate the compiler's as well. Therefore, a model that abstracts away the cells' internal parallelism and pipelining should also hide the synchronization of the cells.

To illustrate this interaction between the internal timing of the cells on systolic algorithm design, let us consider implementing the algorithm in Figure 3-1 on a highly pipelined processor. Suppose the processor has a 3-stage multiplier and a 3-stage adder in its data path. The optimal throughput of an array with such processors is one result every clock cycle. This can be achieved by pipelining the computation, and inserting a 6-word buffer into the x communication path between each pair of cells. A snapshot of the computation of such an array is shown in Figure 3-2.

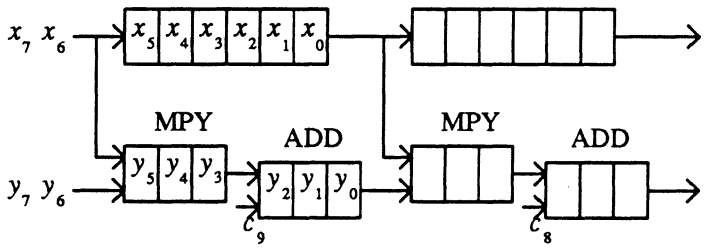


Figure 3-2: Polynomial evaluation on parallel and pipelined processors

In the original algorithm, consecutive cells process consecutive data items concurrently; in the pipelined version, by the time the second cell starts processing the ith data item, the first has already started on the i+6th. Therefore, the decision as to which operations should be executed in parallel must be made with the knowledge of the internal timing of the cells. This is the basic reason why synchronous models in which users specify the concurrent operatiopns do not simplify the task of an optimizing compiler.

The SIMD model of computation offers no assistance to the compiler in translating a program written with a simple machine model into one for pipelined processors. The primitive systolic array model, however, does. The regularity in the computation of systolic arrays can be exploited to allow for this second level of pipelining in the processors. Our cut theorem [36] states that pipelining can be introduced into processors of a systolic array without decreasing the throughput in terms of results per clock, if the data flow through the array is acyclic. This is achieved by adding delays on selected communication paths between the array. If results generated in one clock are used in the next by the same cell, the data flow is still considered cyclic. In case of cyclic data flow through the array, the throughput of the array can only be maintained by interleaving multiple independent computations.

Arrays such as the one in Figure 3-2 can be generated automatically by the use of the cut theorem. However, the boundary conditions that are already complex in the original program translate into even more complex conditions in the optimized program. In the example of the polynomial evaluation, each cell does not start receiving valid data until 6 clocks after its preceding cell has received them. Therefore, 6 sets of fictitious data must be generated on each cell, valid results do not emerge until the 6(m+1)st clock tick, and finally the entire computation takes n+6m+5 clocks. More importantly, the cut theorem can be applied only to simple systolic algorithms, in which all cells repeat the same operation all the time. Any complication such as time-variant computation,

heterogeneous cell programs, or conditional statements would render this technique inapplicable.

In summary, synchronous computation models in which the user completely specifies all timing relationships between different cells are inadequate for high-performance arrays. The reason is that they are hard to program and hard to compile into efficient code. Efficiency can be achieved only in the case of simple programs written using the primitive systolic model. If the primitive systolic model were adopted as the machine model, the versatility of the Warp machine would have been severely reduced.

3.3. Proposed abstraction: asynchronous communication

As shown by the example above, timing information on the internal cell behavior must be used to decide which operations should be executed concurrently on different cells. That is, if the machine abstraction hides the internal complexity of the cells from the user, it must also be responsible to synchronize the computations across the cells. Therefore we propose that the user programs the interaction between cells using asynchronous communication operations: Cells send and receive data to and from their neighbors through dedicated buffers. Only when a cell tries to send data to a full queue or receive from an empty queue will a cell wait for other cells.

Programs are easier to write using asynchronous communication primitives. For example, the program for evaluating polynomials is:

```
/* shift in the coefficients */
c := 0.0;
for i := 0 to m do begin
   Send (R, X, c);
   Receive (L, X, c);
end;

/*  compute the polynomials  */
for i := 0 to n-1 do begin
   Receive (L, X, xdata);
   Receive (L, Y, yin);
   Send (R, X, xdata);
   Send (R, Y, xdata * yin + c);
end;
```

Like the **shift** operation, the **receive** and **send** operations take three arguments: direction, channel used, and a variable name. In a **send** operation, the third parameter can also be an expression. The direction, **L** (left) or **R** (right), and the name of the channel, **X** or **Y**, specify the hardware communication link to be used. In a **receive** operation, the third argument is the variable to which the received value is assigned; in a **send** operation, the third argument is the value sent.

The above cell program is executed by all the cells in the array. The first loop shifts in the coefficients; the second loop computes the polynomials. In the second loop, each cell picks up a pair of **xdata** and **yin**, updates **yin**, and forwards both values to the next cell. By the definition of asynchronous communication, the computation of the second cell is blocked until the first cell sends it the first result. Figure 3-3 shows the early part of the computation of the two cells. This description is simpler and more intuitive, as the asynchronous communication model relieves the user from the task of specifying the exact operations executed concurrently on the cells. It is the compiler's responsibility to synchronize the computation of the cells correctly. This convenience in programming extends to programs using the parallel mode of problem partitioning as well.

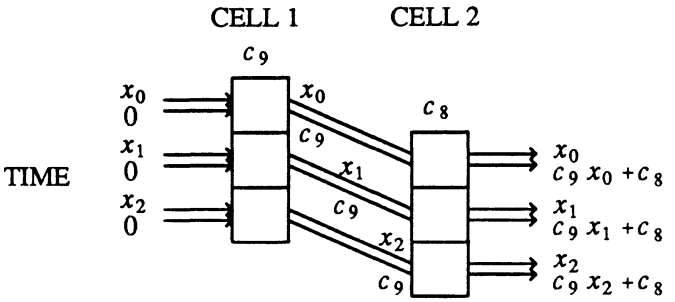


Figure 3-3: Polynomial evaluation using the asynchronous communication model

Cell programs with unidirectional data flow written using the asynchronous communication model can be compiled into efficient array code. Cells in a unidirectional systolic array can be viewed as stages in a pipeline. The strategy used to maximize the throughput of this pipeline is to first minimize the execution time of each cell, then insert necessary buffers between the cells to smooth the flow of data through the pipeline. The use of buffering to improve the throughput has been illustrated by the polynomial evaluation example.

This approach of code optimization is supported by the high-level semantics of the asynchronous communication model. In asynchronous communication, buffering between cells is implicit. This semantics is retained throughout the cell code optimization phase, thus permitting all code motions that do not change the semantics of the computation. The necessary buffering is determined after code optimization.

3.3.1. Effect of parallelism in cells

Let us consider the compilation of the second loop in the program again for cells with a 3-stage multiplier and a 3-stage adder. In a straightforward implementation of the program, a single iteration of the polynomial evaluation loop takes 8 clocks, as illustrated in Figure 3-4.

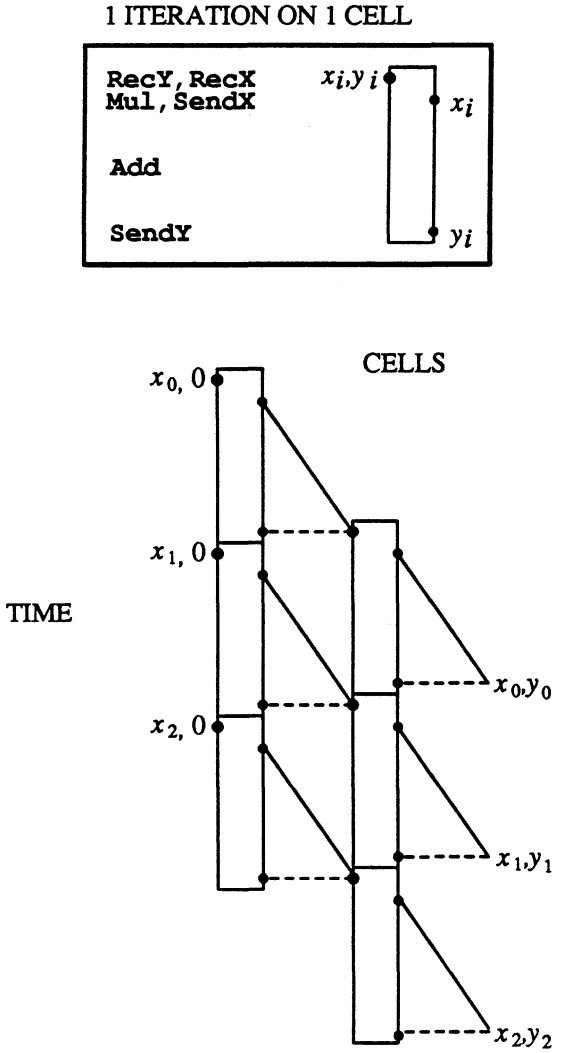


Figure 3-4: Unoptimized execution of polynomial evaluation

The figure contains the microcode for one iteration of the loop, and an illustration of the interaction between cells. The micro-operations **RecX** and **RecY** receive data from the **X** and **Y** queue in the current cell, respectively; and **SendX** and **SendY** send data to the **X** and **Y** queue of the next cell, respectively. The communication activity of each cell is

captured by two time lines, one for each neighbor. The data items received or sent are marked on these lines. The solid lines connecting the time lines of neighboring cells represent data transfers on the **X** channel, whereas the dashed lines represent data transfers on the **Y** channel.

As shown in the figure, the second cell cannot start its computation until the first result is deposited into the **Y** queue. However, once a cell starts, it will not stall again, because of the equal and constant input and output rates of each cell. Therefore, the throughput of the array is one polynomial evaluation every eight clocks.

However, the hardware is capable of delivering a throughput of one result every clock. This maximum throughput can be achieved as follows: We notice that the semantics of the computation remains unchanged if we reorder communication operations on different queue buffers. This observation allows us to perform extensive code motion among the communication operations, and hence the computational operations, to achieve the compact schedule of Figure 3-5.

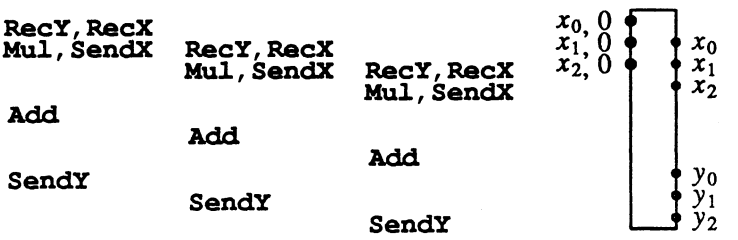


Figure 3-5: Three iterations of polynomial evaluation

The figure shows only three iterations, but this optimal rate of initiating one iteration per clock can be kept up for the entire loop using the scheduling technique described in Chapter 5. The computation of the loop is depicted in Figure 3-6. The only cost of this eight-times speed up is a longer queue between cells. While the original schedule needs a one-word queue between cells, the optimized schedule needs a six-word queue.

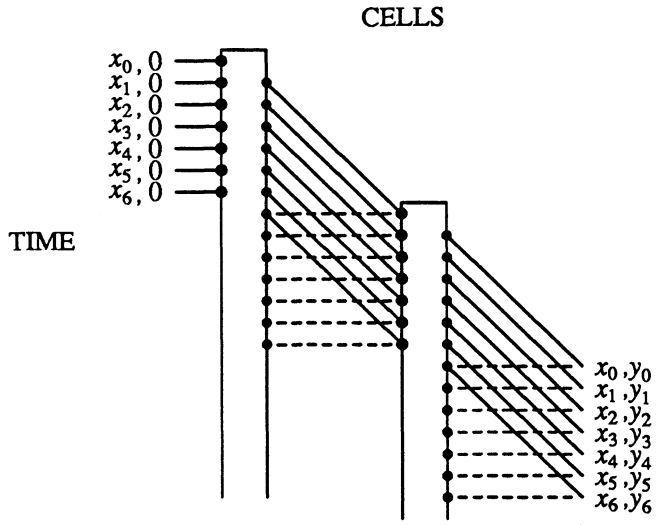


Figure 3-6: Efficient polynomial evaluation

Comparing the two programs in Figures 3-4 and 3-6, we observe the following:

1. In both cases, the second cell must wait 8 clocks before any computation can start, since it needs the first result from the first cell.

2. Once a cell is started, it will never stall again. In both cases, the data consumption rate of a cell equals the data production rate of the preceding cell. The difference, however, is that the rate in the first case is 1 in 8 instructions, whereas that in the second is 1 every instruction. This means that the latency of the first result through the array is the same, but the throughput is improved by 8 times.

3. The computations performed by the two arrays are equivalent. Since the ordering of the receive and send operations on each queue remains unchanged, the same function is performed on the data.

4. The relative ordering of operations on different queues, however, is different. Seven data items are sent on the **X** queue before the first output on the **Y** queue. Since no data is removed from the **X** queue until there is data on the **Y** queue, the **X** queue must be able to buffer up the seven data items. Otherwise, a deadlock situation would occur with the first cell blocked trying to send to a full **X** queue and the second cell blocked waiting for data on the empty **Y** queue. Therefore, relaxing the sequencing constraints between two queues has the effect of increasing the throughput of the system, at the expense of increasing the buffer space requirement along the communication links. The increase in buffer storage is generally insignificant with respect to the payoff in execution speed. Here, the buffer space needs only to be increased by 6 words to achieve an 8-fold speed up.

3.3.2. Scheduling constraints between receives and sends

In general, to use the internal resources of high-performance processors effectively, the sequential ordering of execution in the source program must be relaxed. The approach used in the W2 compiler is to translate the data dependencies within the computation into scheduling constraints, and allow the scheduler to rearrange the code freely so long as the scheduling constraints are satisfied. This approach supports the extensive code motion necessary to use the parallel hardware resources effectively.

As shown in the example above, communication operations in systolic programs must also be reordered to achieve efficient fine-grain parallelism. Fortunately, efficient code can be generated for unidirectional systolic programs by simply analyzing each cell independently and constraining only the ordering of communication operations *within* each cell. These sequencing constraints are represented similarly as data

dependency between computational operations. The uniform representation allows general scheduling techniques to be applied to both communication and computational operations.

3.3.2.1. The problem

Communication operations cannot be arbitrarily reordered, because reordering can alter the semantics of the program as well as introduce deadlock into a program. To illustrate the former, consider the following:

(a)

```
First cell                    Second cell
Send(R,X,1);                  Receive(L,X,c);
Send(R,X,2);                  Receive(L,X,d);
```

(b)

```
First cell                    Second cell
Send(R,X,1);                  Receive(L,X,d);
Send(R,X,2);                  Receive(L,X,c);
```

Programs (a) and (b) are not equivalent, because the values of the variables **c** and **d** in the second cell are interchanged.

To illustrate deadlock, consider the following examples:

(a)

```
First cell                    Second cell
Send(R,X,a);                  Receive(L,X,c);
Receive(R,X,b);               Send(L,X,d);
```

(b)

```
First cell                    Second cell
Receive(R,X,b);               Receive(L,X,c);
Send(R,X,a);                  Send(L,X,d);
```

(c)

```
First cell                    Second cell
Send(R,X,a);                  Send(L,X,d);
Receive(R,X,b);               Receive(L,X,c);
```

The original program (a) is deadlock-free. Reordering the communication operations as in program (b) is illegal because the two cells will be blocked forever waiting for each other's input. While program (b) deadlocks irrespective of the size in the communication buffer, program (c)

deadlocks only if there is not enough buffering on the channels. In this particular example, the cells must have at least a word of buffer on each channel.

We say that the semantics of a program is preserved only if an originally deadlock-free program remains deadlock-free, and that the program computes the same results. Here we answer the following question: given that no data dependency analysis is performed across cells, what are the necessary and sufficient scheduling constraints that must be enforced within each cell to preserve the semantics of the program?

Theoretically, it is possible to allow more code motion by analyzing the data dependency across cells and imposing scheduling constraints that relate computations from different cells. However, such scheduling constraints would greatly complicate the scheduling procedure. Furthermore, since receive and send operations correspond by the order in which they are executed, any conditionally executed receive or send operations would make it impossible to analyze subsequent cell interaction. Therefore, we limit the scope of the analysis to only within a cell program.

3.3.2.2. The analysis

We separate the preservation of semantics of an asynchronous systolic program into two issues: the correctness of the computation (if the program completes), and the avoidance of introducing deadlock into a program. Here we only concentrate on the scheduling constraints pertaining to the interaction between cells; data dependency constraints within the computation in each cell are assumed to be enforced.

First, to ensure correctness, the ordering of operations on the same communication channel must be preserved. That is, we cannot change the order in which the data are sent to a queue, or received from a queue. Since receive and send operations correspond by the order in which they are executed, a pair of receives on the same queue can be permuted only

if the corresponding sends on the sender cell are permuted similarly, and vice versa. Therefore, if code motions on the different cells are not coordinated, the data on each queue must be sent or received in the same order as the original. Conversely, if the ordering of operations on each communication channel is preserved, provided that the program completes, the computation is correct.

On the second issue, let us first analyze the occurrence of a deadlock under a general systolic array model. (Important special cases, such as linear or unidirectional arrays, are given below.) A systolic array is assumed to consist of locally connected cells, where each cell can only communicate with its neighbors. In a deadlock, two or more connected cells must be involved in a circular wait. Each cell involved is waiting for some action by one of the other cells. If the deadlock does not occur in the original program, then the code scheduler must have moved an operation that could have prevented the deadlock past the operation on which the cell is blocked. In other words, the operation on which the cell is blocked *depends* on the execution of some preceding operation, which must be directed at one of the cells involved in the circular wait. Therefore, the key is to first identify all such dependency relationships: a pair of operations such that the first is responsible of unblocking the second. We then insert necessary scheduling constraints to ensure that operations sharing a dependency relationship are not permuted.

If no further information is available on the topology of the array, the direction of data flow, or the size of the data queues, the scheduling constraints are strict. A cell can block either on a receive or send operation, and every receive or send operation can potentially unblock a subsequent receive or send operation. Therefore, the original sequential ordering of all communication operations must be observed.

If the communication buffers are infinite in length, then cells block only on receiving from an empty queue. The only operation that could have unblocked a cell is a send operation. Therefore, the only ordering

that must be preserved is between send operations and subsequent receive operations.

The scheduling constraints can be relaxed in a linear systolic array, provided that there is no feedback from the last cell to the first cell. In a linear array, two and only two cells can be involved in a circular wait: since a cell can only wait for a neighbor one at a time, it is impossible to form a cycle with more than two cells in a linear array. Therefore the unblocking of a communication operation can only depend on receive or send operations from or to *the same cell.* The scheduling constraints above can thus be relaxed as follows: if the queues are infinite in length, send operations to a cell must not be moved below receive operations from the same cell. If the queues are finite, all the sends and receives to and from the same cell must be ordered as before. That is, receive and send operations with the right neighbor are not related to the receive and send operations with the left neighbor.

If the data flow through the array is acyclic, and if the queues are infinite, then no scheduling constraints need to be imposed between communication operations on different channels. This is true of any array topology. The reason is that cells cannot be mutually blocked, and thus there is no possibility of a deadlock. However, if the queues are finite, a cyclic dependency can be formed between the cell and any two of its neighbors. Therefore, as in the general systolic array model, all send and receive operations on every cell must be ordered as in the original program.

To summarize, the ordering of all sends to the same queue, or receives from the same queue, must be preserved. Then, depending on the information available on the topology, the queue model and the data flow, different constraints apply. The information is summarized in Table 3-1. The arrow denotes that if the left operation precedes the right operation in the source program, the ordering must be enforced. The line "send(c) $\rightarrow$ rec(c)" means that the send and receive operation directed to the same cell c must be ordered as in the source program.

General Topology

QUEUE	CYCLIC DATA FLOW	ACYCLIC DATA FLOW
Finite	rec/send → rec/send	rec/send → rec/send
Infinite	send → rec	none

Linear Array

QUEUE	CYCLIC DATA FLOW	ACYCLIC DATA FLOW
Finite	rec/send(c) → rec/send(c)	rec(c) → rec(c) send(c) → send(c)
Infinite	send(c) → rec(c)	none

Table 3-1: Constraints between receive and send operations

3.3.2.3. Implications

Both the finite and infinite queue models permit extensive code motion in the compilation of unidirectional, linear systolic array programs. Since the scheduling of receive and send operations is not constrained, these operations can be scheduled whenever the data are needed, or whenever the results are computed. As in the polynomial evaluation example, the second data set can be received and computed upon in parallel with the first set. It is not necessary to wait for the completion of the first set before starting the second. The send and receive operations are arranged to minimize the computation time on each cell. Provided that sufficient buffering is available between cells, the interaction between cells may only increase the latency for each data set, but not the throughput of the system.

The major difference between the finite and infinite queue models is that in the former, data buffering must be managed explicitly be the compiler. In the finite queue model, because we cannot increase the min-

imum queue size requirement, the receive operations from different queues must be ordered, and so must the send operations. In the polynomial evaluation example, the result of an iteration must be sent out before passing the data (**xdata**) of the next iteration to the next cell. To overlap different iterations, the values of **xdata** from previous iterations must be buffered internally within the cell. The number of data that needs to be buffered exactly equals the increase in minimum queue size in the infinite queue model. Therefore, while the infinite queue model automatically uses the existing communication channel for buffering, the finite queue model requires the buffering to be implemented explicitly. This buffering may be costly especially when the number of operations executed per data item is small.

The infinite queue model has been used successfully in the W2 compiler for the Warp machine. The infinite size model can be adopted because the queues on the Warp cells are quite large; they can hold up to 512 words of data. The increase in queue size requirement due to the allowed code motions is small compared to the size of the queue. The queue size has not posed any problems when cells operate with a fine granularity of parallelism. This infinite queue model is likely to be an important model for other systolic architectures for the reasons below.

First, as discussed above, the infinite queue model can generally allow more code motions than the finite queue model. Second, data buffering is an important part of fine-grain systolic algorithms. It has been used in systolic algorithms to alter the relative speeds of data streams so that data of an operation arrive at the same cell at the same time. Examples include 1-dimensional and 2-dimensional convolutions. We have also shown that buffering is useful in streamlining the computation on machines with parallel and pipelined units. Moreover, a large queue is useful in minimizing the coupling between computations on different cells; this is especially important if the execution time for each data set is data dependent. Therefore, data buffering is likely to be well supported on systolic processors. The infinite queue model uses the data buffering

capability of systolic arrays effectively. Lastly, the increase in the queue size requirement can be controlled. In the current implementation of W2, only communication operations in the same innermost loop are reordered. Code motion is generally performed within a small window of iterations in the loop; the size of the window increases with the degree of internal pipelining and parallelism of the cell. It is possible to further control the increase in buffer size by limiting the code motion of communication operations to within a fixed number of iterations. The analysis and the manipulation of the scheduling constraints are no different from those of computational operations involving references to array variables.

3.3.3. Discussion on the asynchronous communication model

The asynchronous communication semantics allows systolic algorithms to be optimized easily; In this approach, each cell is individually compiled and optimized, and then the necessary buffers are inserted between cells. Results similar to those of the cut theorem can be obtained. One important difference is that while the cut theorem is applicable only to simple, regular computations, the proposed approach applies to general programs.

This approach allows us to obtain efficient code when the following properties are satisfied: the machine has a long queue buffer with respect to the grain size of parallelism, the performance criterion is throughput rather than latency, and the data flow through the array is unidirectional. If any of these properties is violated, more efficient code can probably be achieved if the internal timing of the cells is considered in the computation partitioning phase, or if all the cells in the array are scheduled together.

The effect of cyclic data flow on efficiency depends on the grain size of parallelism. Cyclic data flow has little negative effect on com-

putation of coarse-grain parallelism, but it may induce a significant performance loss in computation of fine-grain parallelism. A common example of the former is domain decomposition, where data is exchanged between neighboring cells at the end of long computation phases. The communication phase is relatively short compared to the computation phase, and it is not essential that it is optimal. In fine-grain parallelism, data sent to other cells are constantly fed back into the same cell. To minimize the time a cell is blocked, we must analyze the dependency across cells to determine the processing time required between each pair of send and receive operations. Besides, Kung and I showed that the internal pipelining and parallelism within a cell cannot be used effectively for a single problem in arrays of cyclic flow [36]. Multiple problems must be interleaved to use the resources in a cell effectively. In the paper, we also showed that many of the problems, for which cyclic algorithms have been proposed, can be solved by a ring or torus architecture. Rings and tori are amenable to similar optimization techniques as arrays of unidirectional data flow. Therefore, the best solution is to rewrite the bidirectional algorithms with fine-grain parallelism as algorithms on rings and tori.

3.4. Hardware and software support

The semantics of asynchronous communication can be directly supported by dynamic flow control hardware. This hardware is provided in the production version of the Warp machine, but not on the prototypes. When a cell tries to read from an empty queue, it is blocked until a data item arrives. Similarly, when a cell tries to write to a full queue of a neighboring cell, the writing cell is blocked until data is removed from the full queue. Only the cell that tries to read from an empty queue or to deposit a data item into a full queue is blocked.

The asynchronous communication model allows us to extend the original computation domain of systolic arrays, and this full generality of this model can be supported only by a dynamic flow control mechanism.

However, many systolic algorithms can be implemented without dynamic flow control hardware. Constructs that are supported on such hardware include iterative statements with compile-time loop bounds, and conditional statements for which the execution time of each branch can be bounded and that the number of receive and send operations on each queue is identical for both branches. The asynchronous communication model does not preclude such programs from being implemented on cells without dynamic flow control support. Static flow control can be implemented after the cell programs have been individually optimized. In fact, this communication model is recommended even for synchronous computation because of its amenability to compiler optimizations. This section discusses the necessary hardware support and presents an efficient algorithm for implementing compile-time flow control.

3.4.1. Minimum hardware support: queues

The minimum hardware support for communication between cells is that data buffering must be provided along the communication path as a queue. While buffering has always been used in systolic computing extensively, a delay element has typically been used. A delay on a data path is an element such that input data into the element emerges as output after some fixed number of clock cycles. Delays are inserted in data paths of many systolic algorithms to modify the speed of a data stream with respect to another [34]. They have also been used to increase the throughput of a system consisting of processors with internal parallelism and pipelining [36]. Even in our earlier example of polynomial evaluation, the queue of the communication links can be implemented by a delay of 8 clocks. In fact, in an earlier design of the Warp cell, the communication buffers were implemented as *programmable delays*.

Although programmable delays have the same first-in first-out property of queues, they are different in that the rate of input must be

tightly coupled with the output. If the input rate into the delay is not one data item per clock, then some of the data item emerging from the delay are invalid and the output must be filtered. A constant delay through the queue means that the timing of data generation must match exactly that of data consumption. Any data dependent control flow in the program cannot be allowed; even if the control flow is data independent, such scheduling constraints complicate the compilation task enormously. Therefore programmable delays are inadequate as communication buffers between programmable processors in an array.

Queues decouple the input process from the output process. Although the average communication rate between two communicating cells must balance, the instantaneous communication rate may be different. A sender can proceed as long as there is room on the queue when it wants to deposit its data; similarly, a receiver can compute on data from the buffer possibly deposited a long time ago until it runs out of data. The degree of coupling between the sender and receiver is dependent on the queue length. A large buffer allows the cells to receive and send data in bursts at different times. The communication buffers on the prototype and the production Warp machines are 128 and 512 words deep, respectively. This long queue length is essential to support the optimizations above, where throughput of the array is improved at the expense of buffer space.

3.4.2. Compile-time flow control

If dynamic flow control is not provided in hardware, as in the case of the Warp prototype machine, then every effort must be made to provide flow control statically. Static flow control means that we determine at compile time when it is safe to execute a receive or send operation, so the generated code does not have to test the queue status at run time. It is expensive to provide dynamic flow control in software on a highly parallel and pipelined machine; explicit testing of queue status on

every queue access would have introduced many more possible branches in the computation, and made it harder to optimize the cell computation. The status of the queues is not even available on the Warp prototype machine. Compile-time flow control is implemented by mapping the asynchronous communication model onto a new computation model, the *skewed* computation model, described below.

3.4.2.1. The skewed computation model

The skewed computation model provides a simple solution to implementing static flow control. In this model, a cell waits to start its computation by the necessary amount of time to guarantee that it would not execute a receive operation before the corresponding send is executed. The delay of a cell relative to the time the preceding cell starts its computation is called the *skew*. Code generated using this model executes as fast, and requires just as much buffering, as a program executing full speed on a machine with dynamic flow control. This section introduces the computation model and its limitations; the next section presents the algorithm for finding this skew.

Let us use a simple example to illustrate the approach. Suppose the following microprogram is executed on two consecutive cells:

```
SendX0
RecX0
RecX1
Add
Add
SendX1
```

The send and receive operations are numbered so we can refer to them individually. Suppose data only flow in one direction: each cell receives data from its left and sends data to its right neighbor.

To ensure that no receive operation overtakes the corresponding send operation, we can take the following steps: first match all receive operations with the send operations, and find the maximum time dif-

ference between all matching pairs of receives and sends. This difference is the minimum skew by which the receiving cell must be delayed with respect to the sender.

Let $\tau_r(n)$ and $\tau_s(n)$ be the time the nth receive or send operation is executed with respect to the beginning of the program, respectively. The minimum skew is given by:

$$\max(\tau_s(n)-\tau_r(n)),\quad \forall 0 \le n < \text{number of receives/sends}$$

Table 3-2 shows the timing of the receive and send operations in the program. The minimum skew, given by the maximum of the differences between the receive and send operations, is three.

n	τ_s	τ_r	$\tau_s-\tau_r$
0	0	1	−1
1	5	2	3
Maximum			3

Table 3-2: Receive/send timing functions and minimum skew

Table 3-3 shows how none of the input operations in the second cell precedes the corresponding send operation in the first cell if the second cell is delayed by a skew of three clocks.

An alternative approach to the skewed computation model is to delay the cells just before the stalling receive and send operations as would happen with dynamic flow control. As illustrated in Table 3-4, the same delay is incurred in this scheme. This approach does not increase the throughput of the machine. The skewed computation model is simpler as it needs only to calculate the total delay necessary and insert them before the computation starts. In the alternative approach, we need to calculate the individual delay for each operation and insert the delays into the code.

The skewed computation model can be used for both homogeneous

Time	First cell	Second cell
0	**SendX$_0$**	
1	**RecX$_0$**	
2	**RecX$_1$**	
3	**Add**	**SendX$_0$**
4	**Add**	**RecX$_0$**
5	**SendX$_1$**	**RecX$_1$**
6		**Add**
7		**Add**
8		**SendX$_1$**

Table 3-3: Two cells executing with minimum skew

Time	First cell	Second cell
0	**SendX$_0$**	**SendX$_0$**
1	**RecX$_0$**	**RecX$_0$**
2	**RecX$_1$**	
3	**Add**	
4	**Add**	
5	**SendX$_1$**	**RecX$_1$**
6		**Add**
7		**Add**
8		**SendX$_1$**

Table 3-4: Dynamic flow control

and heterogeneous programs. The computation of the skew does not assume that the cells are executing the same program. Programs using either the pipelined or parallel style of computation partitioning can be mapped efficiently to this model. Multiple communication paths can also be accommodated as long as the direction of data flow is the same. The skew is given by the maximum of all the minimum skews calculated for each queue. Bidirectional data flow cannot be handled in general using this model. Also, the domain in which compile-time flow control is applicable is inherently restrictive. The restrictions of compile-time flow control and the skewed computation model are explained below.

Data dependent control flow. Compile-time flow control relies on finding a bound on the execution time of the receive and send operations. Therefore, in general, it cannot be implemented if programs exhibit data dependent control flow. Conventional language constructs that have data dependent control flow include WHILE statements, FOR loops with dynamic loop bounds and conditional statements.

Of the data dependent control flow constructs, only a simple form of the conditional statement can be supported: conditional statements that do not contain loops in its branches. The branches of a conditional statement are padded to the same length. As shown in Chapter 6, operations within a conditional statement can be overlapped with operations outside. Therefore, padding the branches to the same length does not imply a significant loss in efficiency. The important point is that now the execution time of the statement remains constant no matter which of its branches are taken. Data can be received or sent within the branches; however, the number of data received or sent on each channel must be the same for both branches. Although we cannot determine at compile time which branch is taken and thus which of the receive/send operations are performed, we can compute a bound on the execution time of the operations. The *minimum* of the ith receive operation in either branch of the conditional statement is considered to be the execution time of the ith receive operation; similarly, the *maximum* of the ith send operation in either branch of the conditional statement is considered to be the execution time of the ith send operation. The minimum is used for receives and maximum is used for sends to ensure that the skew is large enough to handle the worst case.

Bidirectional communication. The skewed computation model simplifies the compile-time control flow problem but it cannot be used on bidirectional programs. Intuitively, the skew delays the receiver with respect to the sender to ensure that no receive operation overtakes the corresponding send operation. With bidirectional flow, it is possible that the second cell must be delayed with respect to the first cell for some pair

of receive and send operations, and the first cell must be delayed with respect to the second for some other pair of operations. Therefore, idle clock cycles cannot be introduced only at the beginning of the computation and must be inserted in the middle of the computation of both cells.

3.4.2.2. Algorithm to find the minimum skew

In the previous section, we explained the skewed computation model by way of a simple straight-line program. We have also explained how conditional statements can be handled just like straight-line code, using the lower and upper bound on the execution times of the receive and send operations. This section describes the algorithm for handling loops.

The complexity of determining the minimum skew in iterative programs depends on the similarity in the control structures in which the matching receive and send statements are nested. Figure 3-7 is an example program that illustrates these two cases. Suppose this

```
Add
Loop 5 times:   RecX0
                RecX1
                Add
Add
Add
Loop 2 times:   SendX0
                SendX1
Add
Add
Loop 2 times:   SendX2
                SendX3
                SendX4
                Add
                Add
Add
```

Figure 3-7: An example program containing loops

microprogram is executed on two cells, and they both receive data from the left and send data to the right.

Table 3-5 gives the timing information on all the receive and send operations. The control structure of the first loop is similar to that of the second loop but not the third. Both the first and second loop contain two I/O statements in each iteration. So, the first and second receive statements are always matched with the first and second send statements, respectively. Since the input rate (2 every 3 clocks) is lower than that of the output (1 every clock), the maximum skew between these loops can be determined by considering only the first iteration. Conversely, if the input rate is higher, only the time difference between the receive/send operations of the last iteration needs to be considered. In the third loop, the number of sends per iteration differs from that of receives in the first loop. Therefore, a receive statement is matched to different send statements in different iterations and all combinations of matches need to be considered in determining the skew. The analysis gets even more complex with nested loops.

Number	τ_s	τ_r	$\tau_s - \tau_r$
0	18	1	17
1	19	2	17
2	20	4	16
3	21	5	16
4	24	7	17
5	25	8	17
6	26	10	16
7	29	11	18
8	30	13	17
9	31	14	17
Maximum			18

Table 3-5: Receive and send timing for program in Figure 3-7

In most programs, the receive and send control constructs are similar since they operate on similar data structures. Furthermore, it is not necessary to derive the exact minimum, a close upper bound will be sufficient. The following mathematical formulation of the problem allows

us to cheaply calculate the minimum skew in the simple cases and its upper bound in the complex ones.

Identifying all the matching pairs of receives and sends is difficult. The key observation is that it is not necessary to match all pairs of receives and sends in the calculation of the minimum skew.

A receive/send *statement* in a loop corresponds to multiple receive/send *operations*. Each receive/send statement is characterized by its own timing function, τ_{r_i} or τ_{s_i}, and an execution set E_{r_i} or E_{s_i}. The timing function maps the ordinal number of a receive or send operation to the clock it is executed. The execution set is the set of ordinal numbers for which the function is valid. For example, the $\mathbf{RecX}_0$ statement is executed 5 times; it is responsible for the 0th, 2nd, 4th, 6th and 8th receive operations which take place in clock ticks 1, 4, 7, 10 and 13 respectively. Its timing function is therefore

$$\tau_{r_0}=1+3n/2,$$

and its execution set is

$$\{n \mid 0 \le n \le 8 \text{ and } n \bmod 2=0\}$$

For each pair of timing functions, τ_{r_i} and τ_{s_j}, we would like to find $\tau_{s_j}-\tau_{r_i}$ for all n that is in the intersection of the execution sets of both functions. The maximum of the differences is the minimum skew.

Finding the exact intersection of the execution sets of two functions may be difficult if their corresponding statements belong to dissimilar control structures. For these cases, instead of using the constraints defining the sets to solve for the intersection completely, we simply use the constraints to bound the difference between the two timing functions.

We characterize each receive or send statement by five vectors of k elements, where k is the number of enclosing loops. Each element of the vector characterizes an enclosing loop, with the first representing the outermost loop. The five vectors are:

$R=[r_1, \ldots, r_k]$: Number of iterations

$M=[m_1, \ldots, m_k]$: Number of receives or sends in one iteration of the loop.

$S=[s_1, \ldots, s_k]$: Ordinal number of the first receive or send in the loop with respect to the enclosing loop.

$L=[l_1, \ldots, s_k]$: Time of execution of one iteration of the loop

$T=[t_1, \ldots, t_k]$: Time to start the first iteration of the loop with respect to the enclosing loop.

For uniformity in notation, the receive or send operations themselves are considered a single-iteration loop. For example, in the program in Figure 3-7, all the vectors describing the operations contain two elements; the first gives information on the enclosing loop, and the second gives information on the statement itself. Therefore, the vector R characterizing r_0 is [5,1], because it is in a 5-iteration loop, and the operation is treated as a single iteration loop. The characteristic vectors for all the receive and send operations in the program are listed in Table 3-6.

Statement	R	M	S	L	T
r_0	[5, 1]	[2, 1]	[0, 0]	[3, 1]	[1, 0]
r_1	[5, 1]	[2, 1]	[0, 1]	[3, 1]	[1, 1]
s_0	[2, 1]	[2, 1]	[0, 0]	[2, 1]	[18, 0]
s_1	[2, 1]	[2, 1]	[0, 1]	[2, 1]	[18, 1]
s_2	[2, 1]	[3, 1]	[4, 0]	[5, 1]	[24, 0]
s_3	[2, 1]	[3, 1]	[4, 1]	[5, 1]	[24, 1]
s_4	[2, 1]	[3, 1]	[4, 2]	[5, 1]	[24, 2]

Table 3-6: Vectors characterizing receive and send operations in Figure 3-7

The timing function of each statement is:

$$\tau(n)=t_1+\left\lfloor \frac{n-s_1}{m_1} \right\rfloor l_1+t_2+\left\lfloor \frac{(n-s_1) \bmod m_1-s_2}{m_2} \right\rfloor l_2+\ldots$$

Every loop nesting of the statement contributes a term to this function. Each term consists of two parts: the starting time of the loop with respect to its enclosing loop, and the time for executing all the iterations that come before the one the nth receive/send is in.

By defining

$$g(j)=\begin{cases} n, & j=0 \\ (g(j-1)-s_{j-1}) \bmod m_{j-1}, & \text{otherwise} \end{cases}$$

we get

$$\begin{aligned}
\tau(n)&=\sum_{j=1}^{k}\left(t_j+\left\lfloor \frac{g(j)-s_j}{m_j} \right\rfloor l_j\right) \\
&=\sum_{j=1}^{k} t_j+\sum_{j=1}^{k}\left(\frac{g(j)-s_j}{m_j} l_j-\frac{(g(j)-s_j) \bmod m_j}{m_j} l_j\right) \\
&=\sum_{j=1}^{k} t_j-\sum_{j=1}^{k}\frac{l_j}{m_j}s_j+\sum_{j=1}^{k}\frac{l_j}{m_j}(g(j)-g(j+1)) \\
&=\sum_{j=1}^{k} t_j-\sum_{j=1}^{k}\frac{l_j}{m_j}s_j+\frac{l_1}{m_1}g(1)+\sum_{j=2}^{k}\left(\frac{l_j}{m_j}-\frac{l_{j-1}}{m_{j-1}}\right)g(j)-\frac{l_k}{m_k}g(k+1)
\end{aligned}$$

The constraints defining the execution set are:

$$\sum_{i=j}^{k} s_i \le g(j) \le (r_j-1)m_j+\sum_{i=j}^{k} s_i$$

The timing functions for the example program and their execution set constraints are given in Table 3-7.

The execution sets of a pair of timing functions can be disjoint, or one may be contained in another or they may overlap partially. The following gives an example in each category from the program in Figure 3-7.

$\tau(n)$	Function	Execution set constraints
r_0	$1+3n/2-n/2 \bmod 2$	$0 \le n \le 8$ and $n \bmod 2=0$
r_1	$1+3n/2-n/2 \bmod 2$	$1 \le n \le 9$ and $n \bmod 2=1$
s_0	$18+n+0n \bmod 2$	$0 \le n \le 2$ and $n \bmod 2=0$
s_1	$18+n+0n \bmod 2$	$1 \le n \le 3$ and $n \bmod 2=1$
s_2	$52/3+5n/3-2(n-4)/3 \bmod 3$	$4 \le n \le 7$ and $(n-4) \bmod 3=0$
s_3	$52/3+5n/3-2(n-4)/3 \bmod 3$	$5 \le n \le 8$ and $(n-4) \bmod 3=1$
s_4	$52/3+5n/3-2(n-4)/3 \bmod 3$	$6 \le n \le 9$ and $(n-4) \bmod 3=2$

Table 3-7: Timing functions for program in Figure 3-7

Disjoint: The execution sets of the functions $\tau_{r_0}(n)$ and $\tau_{s_1}(n)$ do not intersect, since $n \bmod 2=0$ and $n \bmod 2=1$ cannot be satisfied simultaneously. That is, no instance of data items produced by s_1 is read by r_0.

Subset relationship: The execution set of $\tau_{s_0}(n)$ is completely contained in that of $\tau_{r_0}(n)$. That is, all the data items produced by s_0 are read by r_0. The time difference is given by

$$\max \tau_{s_0}(n)-\tau_{r_0}(n) = 17-n/2+n/2 \bmod 2, \text{ where } 0 \le n \le 2 \text{ and } n \bmod 2=0$$
$$\le 17$$

Partially overlapping: The execution sets of $\tau_{r_0}(n)$ and $\tau_{s_4}(n)$ intersect, but are not completely overlapped. Only some of the data produced by s_4 are read by r_0. Here, instead of solving the intersection completely, which may be expensive, we use the constraints defining the sets to bound their time differences:

$$\max \tau_{s_4}(n)-\tau_{r_0}(n) = 52/3-1+(5/3-3/2)n-2(n-4)/3 \bmod 3+n/2 \bmod 2,$$
$$\text{where } 6 \le n \le 8,\ n \bmod 2=1 \text{ and } (n-4) \bmod 3=2$$

$$\le 49/3+1/6\times 8-2/3\times 0+1/2\times 0$$

$$=17+2/3$$

Although the maximum time difference must be determined for every pair of receive and send statements, timing functions corresponding to statements in the same loop share many common terms which need to be computed only once. Also, the branch and bound technique can be applied: bounds on the timing of all the receive/send operations in the same loop can be cheaply obtained to reduce the number of pairs of functions that needs to be evaluated.

Determining the minimum buffer size for the queues is similar to determining the minimum skew. In the minimum skew problem, we define a function for each receive/send statement that maps the ordinal number of the I/O operation to time. In the minimum buffer size problem, we define a function for each receive/send statement that maps time to the number of receive and send operations.

3.4.2.3. Hardware design

Without the need to provide dynamic flow control, the design of the hardware is much simpler. However, there is one important detail: The control for inputting a data item must be provided by the sender. That is, the sender must tag the input data word with a control signal instructing the receiving cell's queue to accept the data. In an earlier prototype of the Warp cell, input data were latched under the microinstruction control of the receiving cell. That is, as the sender presents its data on the communication channel, the receiver issues the control to latch in the input in the same clock cycle. Contrary to our original belief that the coupling between the sender and receiver was no more restrictive than compile-time flow control, it could lead to an intolerable increase in code size.

In the above discussion of compile-time flow control, it is assumed that the control for latching in input is sent with the output data. If the cell receiving the data were to provide the input signals, we need to add a **`LatchX`** operation in the microprogram for every **`SendX`** operation executed by the preceding cell, at exactly the clock the operation takes

place. The simple straight-line program in Figure 3-2 would be as follows:

```
LatchX0,          .
                  .
                  .
          SendX0  .
          RecX0   .
LatchX1,  RecX1   .
          Add     .
          Add     .
          SendX1  .
```

Each line in the program is an micro-instruction; the first column contains the **LatchX** operations to match the **SendX** operations of the output cell, and the second column contains the original program.

Since the input sequence follows the control flow of the sender, each cell actually has to execute two processes: the input process, and the original computation process of its own. If the programs on communicating cells are different, the input process and the cell's own computation process will obviously be different. Even if the cell programs are identical, we may need to delay the cell's computation process with respect to the input process because of flow control. As a result, we may need to merge control constructs from different parts of the program. Merging two identical loops, with an offset between their initiation times, requires loop unrolling and can result in a three-fold increase in code length. Figure 3-8 illustrates this increase in code length when merging two identical loops of n iterations.

If two iterative statements of different lengths are overlapped, then the resulting code size can be of the order of the least common multiple of their lengths. In Figure 3-9, a loop of $3n$ iterations and a 2-instruction loop body is merged with a loop of $2n$ iterations and a 3-instruction loop body. The merged program is a loop of n iterations and a 6-instruction loop body; 6 is the minimum number of clocks before the sequence of operations repeats itself.

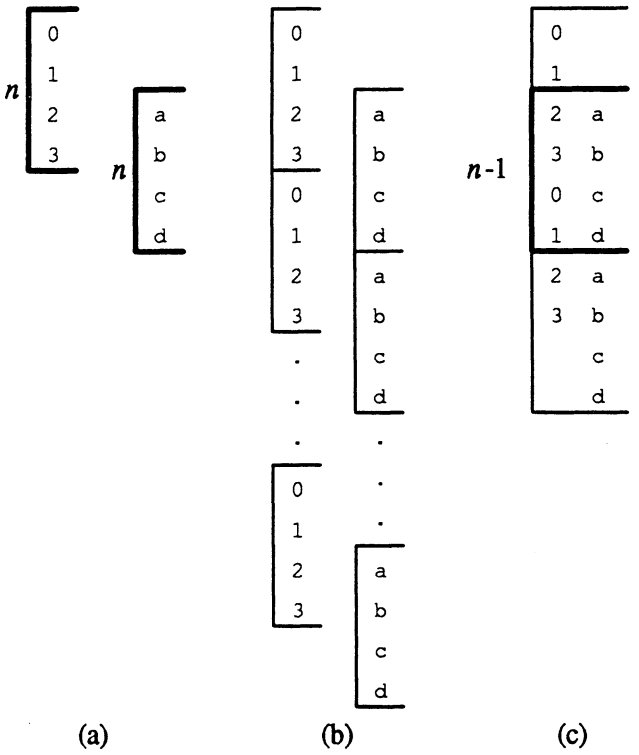


Figure 3-8: Merging two identical iterative processes with an offset (a) original programs, (b) execution trace, and (c) merged program

3.5. Chapter summary

The proposed machine abstraction for a high-performance systolic array is as follows: The user fully controls the partitioning of computation across the array; he sees the machine as an array of simple processors communicating through asynchronous communication primitives. The array level parallelism is exposed to the user because automatic, effective problem decomposition can be achieved only for a limited computation domain presently. On the other hand, the cell level parallelism is hidden because compiler techniques are much more effective than hand coding when it comes to generating microcode for highly parallel and pipelined processors.

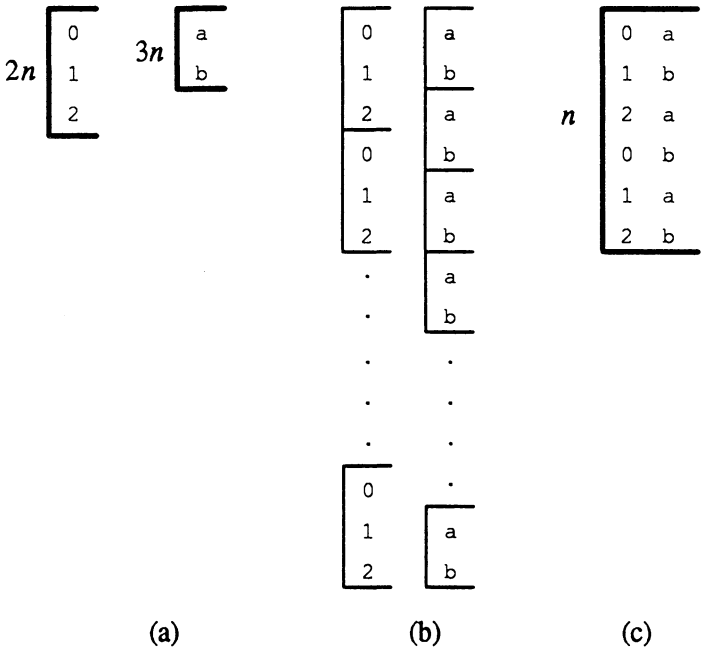


Figure 3-9: Merging two different iterative processes (a) original programs, (b) execution trace, and (c) merged program

The asynchronous communication model is a well-known concept in general computing, but has not been applied to systolic arrays. It is proposed here as the communication model for systolic arrays of high-performance processors not only because of its programmability but also for its efficiency. To achieve efficiency, timing information on the internal cell behavior must be used to decide which operations should be executed concurrently on different cells. That is, if the machine abstraction hides the internal complexity of the cells from the user, it must also be responsible to synchronize the computations across the cells.

The high-level semantics of the asynchronous communication model allows us to relax the sequencing constraints between the communication operations within a cell. Representing only those constraints that must be satisfied in a manner similar to data dependency constraints within a

computation, general scheduling techniques can be used to minimize the execution time on each cell. This approach allows us to generate highly efficient code from complex systolic programs.

The asynchronous communication model is also useful for hardware implementations without direct support for run-time flow control. Computation admissible of compile-time flow control includes all programs with unidirectional data flow and data independent control flow. Programs written using asynchronous communication primitives can be mapped to the skewed computation model where the computation on each cell is delayed with respect to its preceding cell by some predetermined amount of time. The delay, or the skew, can be calculated accurately and quickly for most cases; approximations can be obtained for pathological cases cheaply.

Although the flow control algorithm is no longer used in the compiler for the PC Warp machine, it had served a very important purpose. It was instrumental in the development of the PC Warp architecture. We chose to use the asynchronous communication model on the prototypes because of its programmability and efficiency; it was not a decision driven by hardware. The compile-time flow control made it possible to implement this model on the prototypes, and the success in using the model led to the inclusion of run-time flow control on the PC Warp machine. A large number of applications have been developed on the prototype machine; these programs were ported to the production machine simply by recompiling the programs. The compile-time flow control algorithm may also be used in silicon compilers where custom hardware is built for specific applications.

4

The W2 Language and Compiler

The machine abstraction proposed in the last chapter is supported by the W2 language. W2 is a language in which each cell in the Warp array is individually programmed in a Pascal-like notation, and cells communicate with their left and right neighbors via asynchronous communication primitives. The user controls the array level concurrency, but not the system and cell level concurrency.

The W2 compiler is unconventional in two ways. First, code must be generated for multiple processors cooperating closely to solve a single problem. Specifically, computations on the Warp cells are tightly coupled, and data address computation and host communication must be extracted from the user's program and implemented on the IU and the host. Second, high level language constructs are directly translated into horizontal instruction words. To achieve the global code motions necessary to use the resources effectively, a good global analyzer and scheduler are essential.

This chapter presents an overview of the compiler and prepares the ground for discussion on code scheduling in the next two chapters. It first introduces the W2 language and the high level organization of the compiler, and describes how the system level of parallelism in the Warp architecture is managed by three tightly coupled code generators. The scheduling techniques used in the compiler are based on list scheduling. To introduce the notation used later, the chapter concludes with a description of the list scheduling technique as applied to the problem of scheduling a basic block.

4.1. The W2 language

The W2 language is a simple block-structured language with simple data types. We keep the language simple because we wish to concentrate our effort on efficiency. Shown in Figure 4-1 is an example of a W2 program that performs a 10×10 matrix multiplication. In the matrix multiplication program, each cell computes one column of the result. The program first loads each cell with a column of the second matrix operand, then streams the first matrix in row by row. As each row passes through the array, each cell accumulates the result for a column; at the end of each input row, the cells send the entire row of results to the host.

A W2 program is a *module*; a module has a name, a list of *module parameters* and their type specifications, and one or more *cellprograms*. The module parameters are like the formal parameters of a function; they define the formal names of the input and output to and from the array. An application program on the host invokes a compiled W2 program by a simple function call, supplying as actual parameters variables in the application program.

A cellprogram describes the action of a group of one or more cells. Although the same program is shared by a group of cells, it does not necessarily mean that all the cells execute the same instruction at the same time. As discussed in the previous chapter, computations on dif-

```
module MatrixMultiply (A in, B in, C out)
float A[10,10], B[10,10], C[10,10];

cellprogram (CellId : 0 : 9)
begin
   procedure main();
   begin
      float col[10], row, element, temp;
      int i,j;

      /* first load a column of B in each cell */
      for i := 0 to 9 do begin
         receive (L, X, col[i], B[i,0]);
         for j := 1 to 9 do begin
            receive (L, X, temp, B[i,j]);
            send (R, X, temp);
         end;
         send (R, X, 0.0);
      end;

      /* compute a row of C in each iteration */
      for i := 0 to 9 do begin
         /* each cell computes the dot product
            of its column and same row of A   */
         row := 0.0;
         for j := 0 to 9 do begin
            receive (L, X, element, A[i,j]);
            send (R, X, element);
            row := row + element * col[j];
         end;
         /*  send the result out */
         receive (L, Y, temp, 0.0);
         for j := 0 to 8 do begin
            receive (L, Y, temp, 0.0);
            send (R, Y, temp, C[i,j]);
         end;
         send (R, Y, row, C[i,9]);
      end;
   end;
end
```

Figure 4-1: W2 program for 10×10 matrix multiplication

ferent cells are typically skewed in a pipelined fashion, since a cell cannot start its computation until it has received its input data from the preceding cell. While multiple cellprograms can be specified in the case of the PC machine, only one cellprogram is allowed for the prototype. This restriction is imposed by the lack of address generation capability on the cells. A cellprogram contains definitions of one or more *procedures.*

Within a procedure, four types of statements are supported: the assignment, communication, conditional and iterative statements. The assignment, conditional and iterative statements all have conventional syntax and semantics. However, as explained in Section 3.4.2.1, only limited forms of the control constructs can be supported on the prototype machines because of its lack of dynamic flow control support: Conditional statements must not contain loops in their branches, and the only iterative statements allowed are FOR statements with compile-time loop bounds. Such restrictions are removed for the PC machine.

There are two types of communication statements: *receive* and *send.* They are used to specify the interaction among the cells, as well as between the host and the end cells of the array. The receive and send statements have four parameters: the direction of the channel, the channel name, a local variable and an external (host) variable. The external variable must be a module parameter. (A module parameter cannot be used anywhere else.) A receive statement retrieves a data item from the specified channel and stores it into the local variable. The first cell of the Warp array receives data directly from the host through the IU, and the value is explicitly specified by the external variable; all other cells receive the data transferred in the corresponding send operation of the communicating cell. Similarly, a send statement sends the value of the local variable on the specified channel. In addition, the result from the end cell is stored into the external variable on the host. This external parameter is optional for the send statement. If no external variable is specified, the result is not stored on the host. For example, common data

sent by the host and propagated to all the cells do not need to be stored back on the host.

4.2. Compiler overview

The compiler consists of two major phases: a machine-independent front end and a machine-dependent back end. The front end translates a W2 source program into a machine-independent flow graph and the back end translates the flow graph into code for the Warp cells, the interface unit and the cluster processors. Figure 4-2 is a diagram of the organization of the various components of the compiler.

The steps in the front end include: parsing, local data flow analysis, global data flow analysis, and machine-independent flow graph optimization. The optimizations implemented include common subexpression elimination, constant folding, height reduction, dead code removal, and idempotent operation removal [1].

Global flow analysis provides the information essential for scheduling techniques that overlap operations from different basic blocks. We call these techniques *global scheduling* techniques. They are especially important for heavily pipelined and horizontal processors because of the limited parallelism in a basic block. They rely on accurate global data dependency information. A sophisticated global flow analyzer has been implemented that generates flow information accurate up to the level of individual array elements. It analyzes the data dependency between all array accesses throughout the program, in different basic blocks, different iterations of the same loop, and across different loops. The data dependency information derived is captured by labeled arcs in the flow graph; information relevant to different code optimizations can be easily extracted and obtained from this representation.

The back end has four components: the computation decomposition unit and three code generators, for the Warp array, the interface unit and

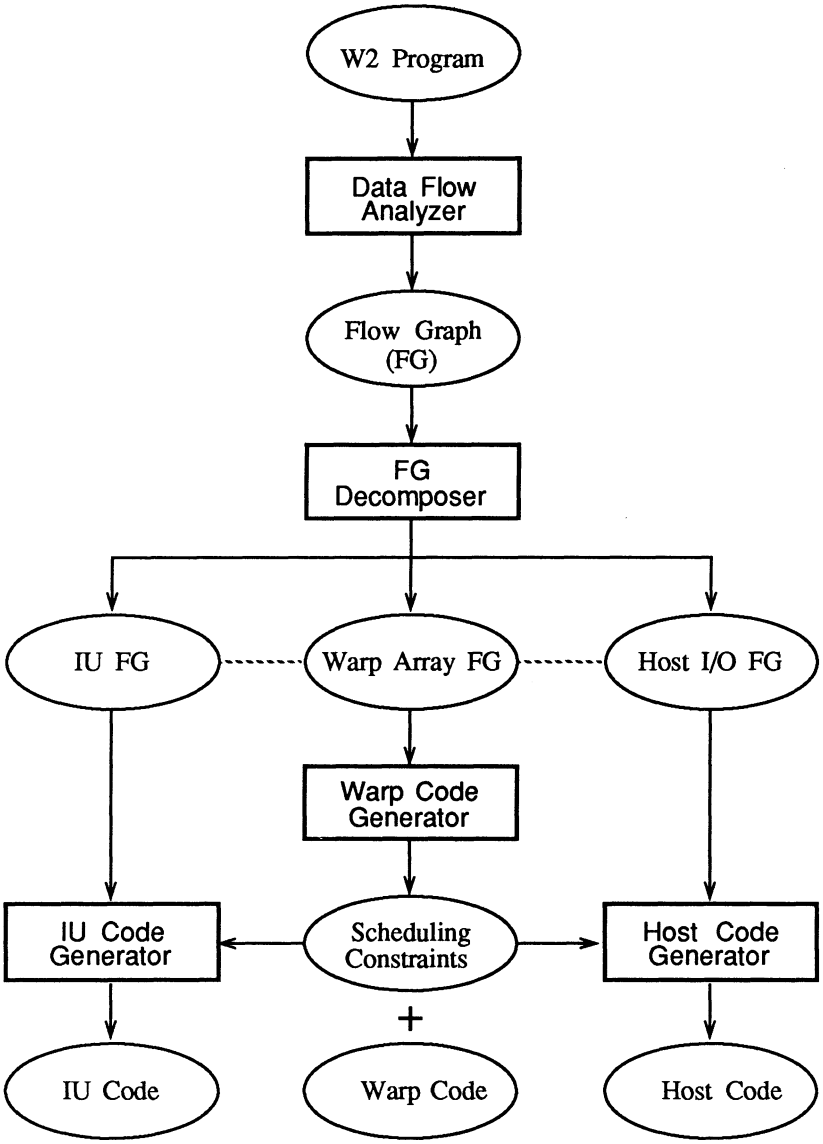


Figure 4-2: Structure of the compiler

the cluster processors. The flow graph generated by the front end is first partitioned into three subgraphs, one for each code generator. For example, a receive statement is decomposed into three parts: the host sends the data to the IU, the IU writes the data into the first cell's queue, and the cell retrieves the data item from the queue and stores it into a local variable. Also, for the prototype machines, local memory addresses determined to be data independent are separated from the flow graph. All array references indexed only by expressions of loop indices are considered data independent and are calculated on the IU. These address calculations in the cell programs are replaced by receive operations from the address queues.

The subgraph for the Warp cells is first fed to the Warp array code generator. From the code generated for the cells, the timing and sequencing information for the input to and output from the array (including addresses on the address queue) are extracted and used by the IU and the host code generators.

The code generator for the Warp cell translates the machine-independent flow graph into microcode for the cells. The process consists of the following steps:

1. Transform the machine-independent flow graph produced by the front end into a machine-dependent flow graph, where generic operators in the former are mapped onto micro-operations. This step is straightforward for Warp because of its orthogonal instruction set and the simple one-to-one correspondence between generic and machine operators.

2. Schedule the operations. The details of this step are described in the following two chapters. Chapter 5 describes the technique of software pipelining for generating highly compact code for innermost loops. Chapter 6 describes an approach called hierarchical reduction which

allows the same scheduling techniques be applied within and across basic blocks.

3. Assign registers. As the assignment of registers is influenced by the schedule, it is performed after code scheduling. The compiler uses a simple greedy algorithm in which registers of maximum lifetimes are assigned first.

4. For the prototype machine, calculate the minimum skew between cells using the algorithm explained in the preceding chapter.

The IU code generator takes its portion of the flow graph and the timing information from the cell code generator and produces microcode for the IU. As dynamic flow control is not implemented on the prototype Warp machines, the IU must supply the data and addresses to the first Warp cell before the latter attempts to remove any data item from the queues. Therefore, the schedule of the cells imposes firm deadlines on the IU. Various optimizations have been implemented on the IU to meet the stringent requirements [28].

Lastly, the host code generator produces C code for the cluster processors and the UNIX host from its portion of the flow graph. The UNIX host transfers data between the host machine and the memory of the cluster processors; the cluster processors transfer data between its memory and the Warp array in the order expected by the cells. An application on the UNIX host interfaces with a program on the Warp array simply by making C function calls.

With this overview of the compiler, we are now ready to present some of the background material necessary for the discussion of the scheduling techniques in the next two chapters.

4.3. Scheduling a basic block

The global scheduling techniques presented in the next two chapters use the basic framework of the list scheduling algorithm. Here, we introduce the list scheduling technique through the simpler problem of scheduling a basic block.

4.3.1. Problem definition

The objects to be scheduled in a basic block are called *micro-operation sequences*. A micro-operation sequence is a series of steps, each of which consists of zero or more micro-operations. It has the following properties:

1. Each step of the sequence may use zero or more units of resources in the machine.

2. A micro-operation sequence is atomic; once a sequence is initiated, it must run to completion without interruption. In Warp, control for the various stages of a pipelined operation is unbuffered; control must be supplied to the particular functional units at different time steps. For example, the destination of the result of an addition must be supplied exactly 7 clock ticks after the operator and the operands are specified.

3. Micro-operation sequences are minimally indivisible. They are as short as possible to maximize flexibility in scheduling. Micro-operations are grouped together as a sequence if and only if the relative timing between operations cannot be changed due to machine characteristics.

There are two kinds of scheduling constraints, resource and precedence constraints.

Resource constraints. Each step in a micro-operation sequence uses

zero or more units of each kind of machine resources. Micro-operation sequences may execute concurrently only if the combined resource requirement for any one step does not exceed the available resources.

Precedence constraints. Data dependencies relating the nodes in the original machine-independent flow graph are mapped onto precedence constraints between the corresponding micro-operation sequences in the machine-dependent flow graph. Associated with each precedence constraint is a minimum delay separating the initiation of the micro-operation sequences. Two factors are involved in computing this delay: the step in which the data is accessed and the latency of the access operation. The latter depends on the machine characteristics; for example, a data item stored into memory may not be available immediately. Therefore, if the ith step of one sequence accesses the data stored in the jth step of another sequence, and that the latency of the store operation is d clock ticks, then the initiation of the first sequence must be delayed with respect to the second by $j-i+d$ clocks.

A basic block is modeled as a graph; the nodes represent the atomic micro-operation sequences, and the edges represent the precedence constraints between the sequences. The schedule of a node is the time the corresponding micro-operation sequence is initiated. The definition of the basic block scheduling problem is:

Let R be the resource configuration vector, where $R(i)$ is the number of units of resource i available in the system configuration. A resource usage vector r is a tuple of the same length where $0 \le r(i) \le R(i)$.

Let $G=(V,E)$ be a directed acyclic graph. Associated with each edge e is a minimum delay $d(e)$. Associated with each vertex v are the following attributes:

- length $l(v)$, and
- a set of resource usage functions $\rho_v(i)$, where

$$\rho_v(i)=\begin{cases} \text{resource usage vector of step } i, & 0\le i<l(v) \\ <0,0,\ldots,0>, & \text{otherwise.} \end{cases}$$

The basic block scheduling problem is to find the shortest schedule $\sigma: V \rightarrow N$ such that

$$\forall e=(u,v)\in E,\ \sigma(v)-\sigma(u)\ge d(e)$$

and

$$\forall i,\ \sum_{v\in V} \rho_v(i-\sigma(v))\le R,$$

where the length of the schedule is defined as $\max_{v\in V}\sigma(v)+l(v)$.

The problem has been shown to be NP-complete [17] by reduction from the discrete processor scheduling problem [8].

4.3.2. List scheduling

Local microcode compaction is the problem of fitting micro-operations in a basic block into as compact a schedule as possible. Various algorithms have been proposed and studied, and this problem has been considered solved [23]. A comprehensive evaluation of the different techniques has been reported in Fisher's dissertation [17]. Fisher recommends list scheduling be used because of its effectiveness and ease of implementation. Although the nodes to be scheduled execute in unit time in Fisher's experiments, list scheduling has also been reported to be effective for operations that consume multiple units of execution time [56].

The original list scheduling algorithm is slightly modified here to handle the multi-step operation sequences. During the scheduling process, the algorithm maintains a "ready" list of nodes whose predecessors have already been scheduled, and thus are ready to be scheduled themselves. The process is iterative. Each iteration consists

of the following steps: select a node from the ready list according to some priority function, schedule it and then update the list. The process terminates when all the nodes have been scheduled.

The algorithm is presented in Figure 4-3. To schedule a node, first compute the earliest time the node can be initiated without violating any precedence constraints with those already scheduled. Starting from this time slot, we test successive time slots to find the first one that satisfies the resource constraints imposed by the nodes scheduled so far. The function *SatisfyResourceConstraint* can potentially be expensive; to test if a slot is eligible to start a sequence, every step in the sequence must be tested for resource conflict. To minimize the cost, the cumulative resource usage for the nodes already scheduled is incrementally compiled. By using a bit-vector array representation as suggested in Fisher's work, testing for conflicts is reduced to a series of logical operations.

4.3.3. Ordering and priority function

List scheduling offers a framework in which to attack the scheduling problem; the algorithm can be adapted and tuned to particular scheduling problems by varying the ordering in which the nodes are scheduled. In this case of scheduling a basic block, the nodes in the flow graph are traversed in a top-down manner, and the priority function used to discriminate between nodes that are ready to be scheduled at the same time is the height of the node, defined below.

By a top-down traversal of the nodes, we mean that those nodes that must execute first are scheduled last. In mapping the operations in the machine-independent flow graph to machine-dependent operations, the direction of the data dependency relationships are reversed when mapped to precedence constraints in the machine-dependent flow graph. That is, when an operation u in the machine-independent flow graph uses the result of the operation v, we say that the machine-operation representing

```
FUNCTION ListSchedule (V, E)
BEGIN
    Ready := root nodes of V;
    Sched := ∅;
    WHILE Ready ≠ ∅ DO
    BEGIN
        v := highest priority node in Ready;
        Lb := SatisfyPrecedenceConstraints (v, Sched, σ);
        σ(v) := SatisfyResourceConstraints (v, Sched, σ, Lb);
        Sched := Sched ∪ {v};
        Ready := Ready−{v}+{u | u ∉ Sched ∧ ∀(w,u) ∈ E, w ∈ Sched};
    END;
    RETURN (σ);
END;

FUNCTION SatisfyPrecedenceConstraint (v, Sched, σ)
BEGIN
    RETURN (max_{u ∈ Sched} σ(u)+d(u,v))
END;

FUNCTION SatisfyResourceConstraint (v, Sched, σ, Lb)
BEGIN
    FOR i := Lb TO ∞ DO
        IF ∀0≤j<l(v), ρ_v(j)+ Σ_{u ∈ Sched} ρ_u(i+j−σ(u)) ≤ R THEN
            RETURN (i);
END;
```

Figure 4-3: List scheduling for a basic block

u precedes that representing *u*. The steps in the micro-operation sequences are inverted, and so is the final schedule generated by the algorithm; the first slot in the schedule actually corresponds to the last microinstruction in the program. By trying to schedule the operations as early as possible in the (inverted) schedule, the operations are actually delayed as much as possible.

The reason for scheduling the original flow graph in an order opposite to the execution order is to minimize the lifetime of register values, the duration from the time a value is stored into a register to the last use of the value. In this reverse ordering, a store into a register is scheduled after all its uses have been scheduled. By placing the store operation as close as possible to the first use of the value, the lifetime of the value is shortened. If scheduling were performed in the order of the execution, a data value would be stored into the register as soon as the value was available regardless of when the register was referenced. The lifetimes of the values and thus the need for registers would be significantly increased.

The priority function used in the W2 compiler is rather conventional. The priority of a node is given by its height, where the height of a node is defined as the longest path from the node to a terminal node. Using this priority function, operations followed by a long chain of computation is started as soon as possible. The rationale is that computation time is governed by the critical path in the graph since the height of the graph is typically much greater than its breadth due to the deep pipelining in the arithmetic units of the machine. In previous experiments with list scheduling, priority functions that also consider the likelihood of resource contention usually performed quite well. In the case of Warp, the resource usage of each micro-operation sequence within a basic block is fairly light, and it is not included in the priority function.

In the next two chapters, the same list scheduling framework is used to schedule operations across basic blocks. However, the ordering and priority function used are quite different to adapt to the particular scheduling problems.

5
Software Pipelining

Pipelining and parallel functional units are common optimization techniques used in high-performance processors. Traditionally, this parallelism internal to the data path of a processor is only available to the microcode programmer, and the problems of minimizing the execution time of the microcode within and across basic blocks are known as local and global compaction, respectively. The development of the global compaction technique, trace scheduling, has led to the introduction of VLIW (very long instruction word) architectures [9, 19, 20, 21]. A VLIW machine is like a horizontally microcoded machine: it consists of parallel functional units, each of which can be independently controlled through dedicated fields in a "very long" instruction. A characteristic distinctive of VLIW architectures is that these long instructions are the machine instructions. There is no additional layer of interpretation where machine instructions are expanded into micro-instructions. A compiler directly generates these long machine instructions from programs written in a high-level language. A VLIW machine generally has an orthogonal instruction set; whereas in a typical horizontally

microcoded engine, complex resource or field conflicts exist between functionally independent operations. Without the code selection problem of horizontally microcoded machines, the scheduling problem of the VLIW machines is significantly simplified.

A Warp cell is a VLIW machine; it has a wide instruction format and consists of multiple pipelined functional units. For example, the data path can support the overlapped execution of fourteen multiply-and-accumulate steps. In a single instruction, the Warp processor can initiate two floating-point operations, read and write a word from and into memory, input and output two data words from and to neighboring cells, branch to any location in program memory, and receive and forward two data addresses in the case of the prototype machine, and perform two integer operations in the case of the PC machine.

The VLIW machine organization is confined to the processor level architecture. The Warp machine is an MIMD (multiple instruction, multiple data) array: each cell has its own sequencing control and executes its own program. The degree of parallelism in each cell is limited to several different (pipelined) functional units. One control, or branch, operation per instruction is sufficient to balance the number of arithmetic operations that can be executed in parallel.

It is interesting to contrast our approach with the large-scale VLIW approach, where the processors contain tens of functional units and a multi-way branching capability [18]. The modular design of Warp offers a path to a scalable, high-performance system by interconnecting large number of processors together. With the support of high-bandwidth and low-latency communication, the Warp processors can be programmed to cooperate with a fine granularity of parallelism, much like the functional units in a large-scale VLIW machine. However, the individual sequencing control on the cells enable them to operate independently. This additional capability to implement coarse-grain parallel programs greatly adds to the versatility of the Warp system.

In this chapter, we propose an alternative approach to trace scheduling for scheduling VLIW machine code. We use *software pipelining* and *hierarchical reduction.* A major distinction between our approach and trace scheduling is that we exploit the characteristics of different control constructs using different techniques. *Software pipelining* is a scheduling technique that exploits the repetitive nature of loops to generate highly efficient code [44, 47, 56]. Iterative constructs deserve special attention because most computation time is spent in innermost loops, especially in numerical processing. Conditional statements, on the other hand, are difficult to optimize because less information is known about their branch probability, so we concentrate on preventing conditional statements from serializing the execution of other operations. In particular, the presence of conditional statements in a loop must not prevent the loop from being software pipelined. We propose a *hierarchical reduction* scheme whereby entire control constructs are reduced to an object similar to an operation in a basic block. This allows techniques defined for scheduling basic blocks to be applicable across basic blocks. This chapter describes software pipelining and the next chapter describes hierarchical reduction.

In software pipelining, iterations of a loop in the source program are continuously initiated at constant intervals, before the preceding iterations complete. The advantage of software pipelining is that optimal performance can be achieved with a compact object code. Software pipelining was originally developed for hand microcoding vector operations. The technique has been used in the compilers for the ESL polycyclic machine [47], Cydrome's Cydra 5 [10], and the FPS-164 computer [56]. The problem of finding the optimal schedule has been shown to be NP-complete. The polycyclic and Cydra compilers depend on expensive specialized hardware support to simplify the scheduling problem; the FPS compiler uses scheduling heuristics, but limits the application of software pipelining to a restricted class of loops.

This work extends the previous results of software pipelining in two

ways. First, we show that expensive hardware support is not necessary for software pipelining by improved scheduling heuristics and a new optimization called *modulo variable expansion.* The latter has some of the functionality of the specialized hardware proposed in the polycyclic machine and achieves similar performance. Empirical results indicate that the proposed scheduling heuristics often produce near-optimal schedules.

Second, software pipelining has previously been applied only to loops with straight-line code bodies. The hierarchical reduction scheme described in the next chapter makes software pipelining applicable to all loops. In particular, all innermost loops, including those containing conditional statements, can be software pipelined. If the number of iterations in the innermost loop is small, we can software pipeline the second level loop as well to obtain the full benefits of this technique.

Software pipelining, as addressed here, is the problem of scheduling operations within an iteration, such that the iterations can be pipelined to yield optimal throughput. The assumed machine model is that the machine contains one or more, possibly different, possibly pipelined, functional units. Software pipelining has also been studied under different contexts. The software pipelining algorithms proposed by Su et al. [52, 53], and Aiken and Nicolau [2], assume that the schedules for the iterations are given and cannot be changed. This constraint was relaxed in Aiken and Nicolau's subsequent paper in which they studied the problem of generating an optimal software pipelined schedule for a machine with infinitely many resources. Ebcioglu also studied a similar problem, and suggested an algorithm for software pipelining loops with conditional statements [14]. Finally, Weiss and Smith compared the results of using loop unrolling and software pipelining to generate scalar code for the Cray-1S architecture [57]. However, their software pipelining algorithm only overlaps the computation from at most two iterations. The unfavorable results obtained for software pipelining can be attributed to the particular algorithm rather than the software pipelining approach.

This chapter concentrates on software pipelining loops whose body consists of a single basic block. (Other cases are discussed in the next chapter). We first introduce the software pipelining technique, present the mathematical formulation of the scheduling problem, and describe the scheduling heuristics. We then introduce the modulo variable expansion optimization. The scheduling algorithm is derived from the techniques used for the FPS and the polycyclic machines; a detailed comparison between the techniques used is presented at the end of the chapter. In addition, this chapter includes a study of the specialized interconnection proposed for the polycyclic machine, evaluating the tradeoff between hardware and software complexity. Comparison with trace scheduling is presented in the next chapter after the description of hierarchical reduction.

5.1. Introduction to software pipelining

The effect of software pipelining can be illustrated by the following example:

```
FOR i := 1 TO 9 DO
BEGIN
    Receive (L, X, a);
    Send (R, X, a+b);
END;
```

Consider compiling this program into code for the Warp cell. Although the processor can support the initiation of one floating-point multiplication and one floating-point addition every instruction, the latency for each operation is long. At the machine instruction level, floating-point arithmetic operations have a seven-instruction, or seven-clock, latency; this latency includes a two-clock delay in fetching data from the multi-ported register file, and a five-clock floating-point operation. Using the same notation as in Chapter 3, the most compact instruction sequence possible for a single iteration of this loop is given in Figure 5-1. The schedule is sparse due to the heavy pipelining in the data path. If we simply iterate this schedule, the throughput of the loop is only 1 iteration

```
RecX.
Add.    ; a+b
.       ; 7-stage pipelined adder
.
.
.
.
SendX.  ; Send result from adder out
```

Figure 5-1: Object code for one iteration in example program

every 8 clock ticks, and no resources are used more than 1/8th of the time.

Trace scheduling relies on unrolling to improve the throughput of a loop [19]. The body of the loop is unwound some number of times and code compaction is performed on the unrolled source code. Unfortunately, there is no clear criterion to determine the suitable degree of unrolling. The utilization almost always improves as more iterations are unrolled; however, the problem size and the resulting code size increase likewise. Suppose the loop body of the example is unrolled 8 times, the optimal schedule of the body of the unrolled loop is shown in Figure 5-2.

```
L:RecX.
  Add, RecX.
       Add, RecX.
            Add, RecX.
                 Add, RecX.
                      Add, RecX.
  SendX,                  Add, RecX.
       SendX,                  Add.
            SendX.
                 SendX.
                      SendX.
                           SendX.
                                SendX.
  CJump L,                        SendX.
```

Figure 5-2: Optimal schedule for 8 iterations

This microcode sequence assumes that the number of iterations is divisible by 8. Each row in the figure corresponds to operations in an instruction, and each column corresponds to the computation of one iteration of the loop in the source program. The operation **CJump L** branches back to label **L** if there are more iterations to execute. In Warp, one branch operation can execute in parallel with other micro-operations. Unrolling the loop 8 times improves the throughput to 8 iterations every 15 clocks. For this program, unrolling the loop k times increases the utilization of the resources to $k/k+7$.

In software pipelining, we do not wait till the completion of an iteration before initiating the next iteration. The iterations in the loop are executed in a pipelined fashion, with multiple iterations executing at the same time in the steady state. Let us consider the same example again. It is obvious from Figure 5-2 that the rate of initiating one iteration every instruction can be kept up until we run out of iterations. Intuitively, the eight instructions in an iteration of the loop can be viewed as an 8-stage pipeline that accepts a new iteration every clock (Figure 5-3). In the

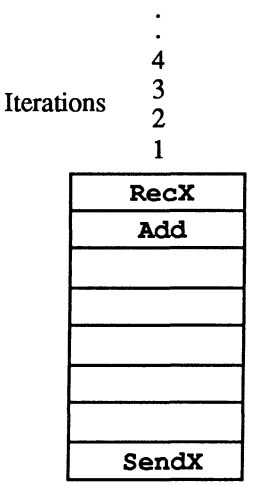


Figure 5-3: The software pipeline

steady state, 8 consecutive iterations of the loop are executed in parallel,

with each one in a different stage of its processing. It is this analogy to a hardware pipeline that software pipelining derives its name.

The schedule for the software pipelined loop can be succinctly represented by a program that is only about twice as long as the program for a single iteration (Figure 5-4). The program in the figure assumes that there are at least eight iterations in the loop. Instructions 1 to 7 are called the *prolog*, in which more and more iterations of the loop start executing. The *steady state* is reached in clock 7, and is repeated until all iterations have been initiated. In the steady state, eight iterations are in progress at the same time, with one iteration starting up and one finishing off every instruction executed. On leaving the steady state, the iterations currently in progress are completed in the *epilog*, instructions 9 to 15. This program achieves the *optimal* computation time by completing n iterations in n+7 clock ticks, where n is the number of iterations in the loop. After a latency of eight clock ticks, the iterations are executed at the optimal throughput of one iteration every clock.

```
        RecX,
        Add,    RecX.
        Add,    RecX.
        Add,    RecX.
        Add,    RecX.
        Add,    RecX.
        Add,    RecX.
L:      SendX,  Add,  RecX,  CJump L.
        SendX,  Add.
        SendX.
        SendX.
        SendX.
        SendX.
        SendX.
        SendX.
```

Figure 5-4: Program of a software pipelined loop

Software pipelining is unique in that the pipeline stages in the functional units in the data path are not emptied at iteration boundaries; the

pipelines are filled and drained only on entering and exiting the loop. The significance of software pipelining are: (1) optimal throughput is achievable, and (2) the code generated is extremely compact.

5.2. The scheduling problem

Software pipelining was originally introduced for scheduling hardware pipelines, and the problem was formulated as inserting delays between hardware units to increase the overall throughput of the system [44]. New input is accepted by the hardware pipeline at regular periodic intervals. The software analog of this approach is to derive a same schedule for each iteration, then initiate and stagger the iterations at periodic intervals. The repeating sequence of states are captured by a loop of microinstructions, one for each state. To minimize the number of states, and thus the size of the microcode, Rau and Glaeser [47] suggested keeping the initiation intervals between every pair of iterations the same. Therefore, to generate the object code for a software pipelined loop, we need the schedule of an iteration and the *iteration initiation interval* [47], the interval at which the iterations of the loop are initiated. In the example above, the schedule of an iteration is given in Figure 5-1, and the initiation interval is one.

The objective of software pipelining is to find a schedule for an iteration that can be initiated at shortest intervals. The following describes the scheduling constraints of the problem and presents the mathematical formulation of the problem.

5.2.1. Scheduling constraints

As in the scheduling problem for basic blocks, there are two kinds of scheduling constraints: resource and precedence constraints. The difference, however, is that both sets of constraints are defined in terms of the initiation interval of the schedule.

Resource constraints. If iterations in a software pipelined loop are initiated every sth clocks, then every sth instruction in the schedule of an iteration is executed simultaneously, one from a different iteration. Therefore, the total resource requirement of every sth instruction cannot exceed the available limit. Using the terminology defined in Chapter 4, we represent the resources required clock i by a *modulo resource usage function*, $\overline{\rho}^s$:

$$\overline{\rho}^s(i) = \sum_{k \in Z} \rho(i+ks).$$

The resource constraint is thus:

$$\forall\, 0 \le i < s, \quad \sum_{v \in V} \overline{\rho}_v^s(i - \sigma(v) \bmod s) \le R.$$

The overlapped resource usage from different iterations is illustrated in Figure 5-5(a). The resource usage for each iteration is represented by

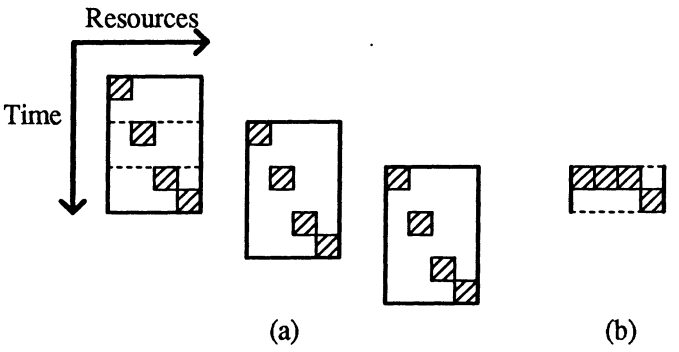


Figure 5-5: (a) Overlapped resource usage and (b) modulo resource usage function

a reservation table. The entry in column i and row j represents the use of the ith resource in step j. A shaded entry corresponds to the use of the corresponding resource. (In this representation, the machine model can have only one unit of each kind of resources. This is only a limitation to simplify the illustration; the mathematical formulation of the problem

and the proposed algorithm support the presence of multiple units of identical functional units.) Figure 5-5(b) shows the resource usage in the steady state. This modulo resource usage table can be derived by folding the resource usage of each iteration into a table of size *s*, the initiation interval.

Precedence constraints. As iterations of a loop are overlapped in software pipelining, global data dependencies between operations from different iterations must be considered. Consider the following example:

```
FOR i := 1 to 100 DO
BEGIN
    a := a + 1.0;
END;
```

The minimum delays between the read and write operations of the variable **a**, assuming the machine characteristics of the Warp cell, are depicted in Figure 5-6(a). Due to the pipelined addition, the write opera-

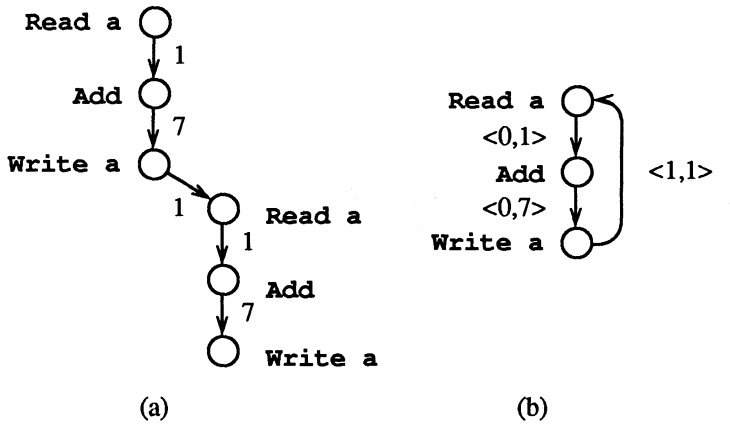


Figure 5-6: (a) Delays between operations from two iterations, and (b) flow graph representation

tion must be scheduled at least 8 clocks after the read operation of the current iteration. The read operation, however, must wait for the write operation from the previous iteration to complete. (Please note that here we explain the algorithm as if the nodes are scheduled in the order they

are executed. In reality, the graph is inverted, and operations that must execute first are scheduled last, as explained in Section 4.3.3 of Chapter 4. As this reversal in scheduling order is orthogonal to the software pipelining technique, we ignore this reversal in the explanation here to make it more intuitive.)

The precedence constraints are captured by labeling the edges not with a single delay as in the basic block scheduling problem, but by a tuple: the minimum number of iterations separating the related instances of the nodes and the minimum delay between them. When we say that the minimum iteration difference on an edge (u,v) is p and the delay is d, it means that the node v must execute d or more clocks after node u from the p th previous iteration. That is,

$$\sigma(v)-(\sigma(u)-s\cdot p(u,v))\geq d(u,v), \quad \text{or} \quad \sigma(v)-\sigma(u)\geq d(u,v)-s\cdot p(u,v),$$

where s is the initiation interval. Since a node cannot depend on a value from a future iteration, the minimum iteration difference is always non-negative. The iteration difference for an intra-iteration, or loop-independent, dependency is 0, meaning that the node v must follow node u in the same iteration. The iteration difference between an inter-iteration, or loop-carried, dependency must be positive. The flow graph representing the above example is shown in Figure 5-6(b). As shown in the figure, cycles can exist in the graph. The existence of cycles in the graph complicates the scheduling problem significantly; more will be discussed later.

The minimum iteration difference is extracted from the information obtained by the global data dependency analyzer. The following example illustrates the different kinds of dependency:

```
FOR i := 1 to 100 DO
BEGIN
    a[i] := a[i] + 1.0;
    c[i] := c[i-2] + 1.0;
    d[i] := d[3] + 1.0;
END
```

The assignments to variable `a[i]` in different iterations are independent, so there is no inter-iteration precedence constraint between these operations. The variable `c[i-2]` refers to the value stored into the array two iterations ago; therefore, the iteration difference is 2. And in the case of the variable `d`, the iteration difference between the related instances of the nodes is not constant. The worst case is assumed and we label the precedence constraint with the minimum separation of one iteration.

5.2.2. Definition and complexity of problem

The definition of the software pipelining problem is as follows:

Let R be the resource configuration vector, where $R(i)$ is the number of units of resource i available in the system configuration. A resource usage vector r is a tuple of the same length where $0 \le r(i) \le R(i)$. Let $G=(V,E)$ be a directed graph. Associated with each vertex v are the following attributes:

- length $l(v)$, and
- a resource usage function ρ_v, where

$$\rho_v(i)=\begin{cases} \text{resource usage vector for step } i, & 0 \le i < l(v) \\ <0,0,\ldots,0> & \text{otherwise} \end{cases}$$

The number of jth resource used in step i is denoted by $\rho_v(i,j)$.

Associated with each directed edge e are two quantities:

- the minimum iteration difference by which the related nodes are separated: $p(e)$, and
- the minimum delay between the nodes, $d(e)$.

The problem is to find the minimum initiation interval s and a schedule $\sigma: V \rightarrow N$ such that the precedence and resource constraints are satisfied, i.e.,

$$\forall e=(u,v) \in E,\ \sigma(v)-\sigma(u) \ge d(e)-s \cdot p(e)$$

and

$$\forall 0 \le i < s, \quad \sum_{v \in V} \overline{\rho}_v^s (i - \sigma(v) \bmod s) \le R.$$

The problem of whether a schedule can be found for a given initiation interval is NP-complete; this can be shown by reduction from the resource constrained scheduling problem [27]. In the resource constrained scheduling problem, the tasks do not have precedence constraints and they all take unit execution time. To ask if a schedule can be found in n clocks can be reduced to asking if a schedule can be software pipelined with an initiation interval n.

5.3. Scheduling algorithm

There have been two approaches in response to the complexity of this problem: (1) change the architecture, and thus the characteristics of the constraints, so that the problem is no longer NP-complete, and (2) use software heuristics. The first approach is used in the polycyclic architecture; a specialized crossbar is used to make optimizing a subset of loops tractable. The second approach is used in the FPS compiler. We also use software heuristics; we have improved the heuristics and extended the applicability of the technique to include all loops. Comparison of our algorithm with different techniques will be presented after the description of the algorithm.

The software pipelining algorithm is complicated because of two reasons. First, the scheduling constraints are defined in terms of the initiation interval, and this makes finding an approximate solution to this NP-complete problem difficult. Since computing the minimum initiation interval is NP-complete, a standard approach is to first schedule the code using heuristics, and then determine the initiation interval permitted by the schedule. However, since the scheduling constraints are a function of the initiation interval, if the initiation interval is not known at scheduling time, the schedule produced is unlikely to permit a good initiation interval.

To resolve this circularity, the FPS compiler uses an iterative approach: first establish a lower and an upper bound on the initiation interval, then use binary search to find the smallest initiation interval for which a schedule can be found [56]. (The length of a locally compacted iteration can serve as an upper bound; the calculation of a lower bound is described below). We also use an iterative approach, but we use linear instead of binary search. That is, we try to find a schedule using the lower bound of the initiation interval as the target interval. We iterate this process if we fail to find such a schedule by increasing the target initiation interval by one clock tick at a time. The rationale is as follows: Although the probability that a schedule can be found generally increases with the value of the initiation interval, schedulability is not monotonic, as explained below. Especially since empirical results show that, in the case of Warp, a schedule meeting the lower bound can often be found, sequential search is preferred.

The second cause for the complexity of the scheduling algorithm is the presence of cycles in the graph. For a given initiation interval, an *acyclic* graph can be scheduled using the list scheduling algorithm described in the previous chapter, substituting the resource function with the modulo resource usage function. Cycles in the flow graph, however, make designing an effective non-backtracking scheduling algorithm difficult. The algorithm proposed here first finds the *strongly connected components* of the graph, schedules the individual strongly connected components using a different set of heuristics, and finally schedules the components themselves using the list scheduling algorithm for acyclic graphs.

In the following, we discuss in detail the bounds on the initiation interval, the algorithm for acyclic graphs, and finally, the algorithm for cyclic graphs.

5.3.1. Bounds on the initiation interval

A lower bound on the initiation interval can be derived from the resource constraints for all graphs; if the graph is cyclic, a bound is further imposed on the initiation interval by the precedence constraints. As shown in Chapter 7, empirical results show that the bound obtained using the formulae below is often strict.

Resource constraints. If an iteration is initiated every s clocks, then the total number of resource units available in s clocks must at least cover the resource requirement of one iteration. Therefore, the bound on the initiation interval due to resource considerations is the maximum of the total number of times each resource is used divided by the available units per clock:

$$S_R = \max_k \left\lceil \frac{\sum_{v \in V, 0 \le i < l(v)} \rho_v(i,k)}{R(k)} \right\rceil .$$

To see that the bound is not necessarily tight even if there are no precedence constraints between the nodes, consider a loop whose body consists of two nodes with resource functions:

Resource conflicts preclude such a loop from being pipelined with initiation interval S_R (=2).

Precedence constraints. Cycles in precedence constraints impose a minimum distance between operations from different iterations. The initiation interval must be large enough for such delays to be observed. For example, the precedence constraints in Figure 5-6 impose a delay of 8 clock ticks between read operations from consecutive iterations. That is, the initiation interval has to be no smaller than 8 clocks. We define the minimum delay and minimum iteration difference of a path to be the sum of the minimum delays and minimum iteration differences of the

edges in the path, respectively. Let s be the initiation interval, and c be a cycle in the graph. Since

$$\sigma(v)-\sigma(u)\geq d(e)-s\cdot p(e)$$

we get:

$$d(c)-s\cdot p(c)\leq 0.$$

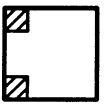
We note that if $p(c)=0$, then $d(c)$ is necessarily less than 0 by the definition of a legal computation. Therefore, the bound on the initiation interval due to precedence considerations is

$$S_E=\max_c \frac{\lceil d(c) \rceil}{p(c)}, \quad \forall \text{ cycle } c \text{ whose } p(c)\neq 0.$$

The lower bound of the initiation interval is given by the maximum of the lower bounds established by resource and precedence considerations.

We can also bound the initiation interval *from above* by list scheduling one iteration of the loop. The length of the schedule serves as an upper bound of the initiation interval. Unless a schedule can be found for an initiation interval shorter than the schedule of an iteration, there is no advantage to software pipelining. We call this upper bound $S_{\max}$.

The probability of finding a schedule generally increases with the value of the initiation interval. However, schedulability is not monotonic. For example, a node with the following resource usage can be initiated every 2, but not 3, clocks.



Especially since empirical results reported in Chapter 7 show that the lower bound can almost always be met, sequential search is more preferable than binary search.

5.3.2. Scheduling an acyclic graph

For acyclic graphs, the initiation interval s is bounded from below by only S_R. We try to find a software pipeline schedule starting with this lower bound. The algorithm we use to schedule an acyclic graph for a target initiation interval is the same as that used in the FPS compiler, which itself is derived from the list scheduling algorithm used in basic block scheduling [17]. The difference is that the modulo resource usage function is used in computing resource conflicts. By the definition of modulo resource usage, if we cannot initiate a node in s consecutive clock ticks due to resource conflicts, it will not fit in the current schedule. When this happens, we abort the attempt to find a schedule for the given initiation interval and repeat the scheduling process with a greater interval value. If no schedule is found for an initiation interval value less than S_{max}, simple list scheduling is used.

An example of an acyclic flow graph is shown in Figure 5-7(a).

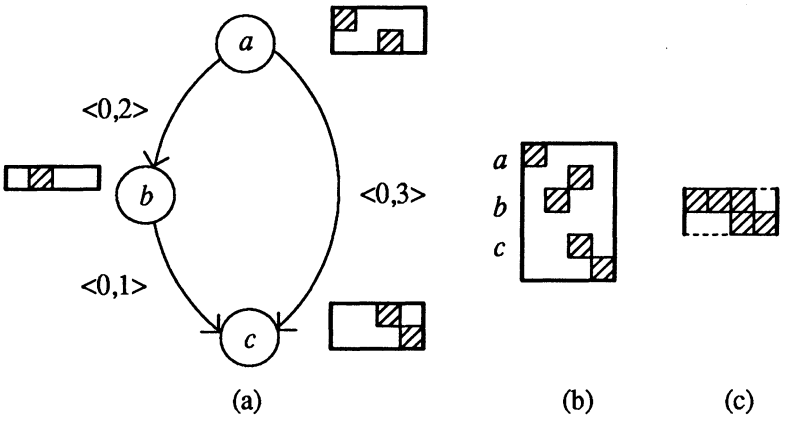


Figure 5-7: Scheduling an acyclic graph: (a) an example flow graph, (b) resource usage of an iteration, and (c) modulo resource usage

Since the third resource is used twice in an iteration, the initiation interval is at least two clocks. We first attempt to schedule the loop for a

target initiation interval of two clock ticks. By following the topologically ordering of the precedence constraints, we schedule node *a* in clock 0, then node *b* in clock 2. Although precedence constraints permit node *c* to be scheduled in clock 3, execution of the node is delayed until clock 4 to avoid creating a conflict in the modulo resource reservation table. The resource usage of a single iteration and the modulo resource reservation table are shown in Figure 5-7(b) and (c), respectively. The rows in the reservation table are labeled with the nodes that are initiated in the corresponding clock ticks. This example shows that a locally optimal schedule for a single iteration does not necessarily result in a globally optimal pipelined schedule for the loop.

The algorithm for software pipelining an acyclic graph is given in Figure 5-8. The function *ListSchedule* is now modified from the definition in Chapter 4 to find a schedule for a basic graph within a loop, for a given initiation interval *s*. Scheduling a basic block not nested any loops is a special case. This can be achieved by specifying the initiation interval to be infinity. The function *SatisfyResourceConstraints* searches for a slot to initiate the node between the given lower and upper bounds. If no such slot is found, the function reports that the attempt to schedule the node is unsuccessful. *ListSchedule* is called iteratively with increasing values of target initiation intervals until a schedule is found or until the upper bound is met.

FUNCTION *SchedAcyclicGraph* (V, E)
BEGIN
$S_R := \max_k \left\lceil \frac{\sum_{v \in V, 0 \le i < l(v)} \rho_v(i,k)}{R(k)} \right\rceil$;
S_{max} := length of *ListSchedule* (V, E, ∞);
FOR $s := S_R$ TO S_{max} DO
BEGIN
σ := *ListSchedule* (V, E, s);
IF $\sigma \neq \perp$ THEN RETURN $(<\sigma, s>)$;
END;

```
    RETURN (<ListSchedule (V, E, ∞), S_max>);
END;

FUNCTION ListSchedule (V, E, s)
BEGIN
    Ready := root nodes of V;
    Sched := ∅;
    WHILE Ready ≠ ∅ DO
    BEGIN
        v := highest priority node in Ready;
        Lb := SatisfyPrecedenceConstraints (v, Sched, σ, s);
        σ(v) := SatisfyResourceConstraints (v, Sched, σ, s, Lb, Lb+s−1);
        IF σ(v) = ⊥ THEN RETURN (⊥);
        Sched := Sched ∪ {v};
        Ready := Ready−{v}+{u | u ∉ Sched ∧ ∀(w,u) ∈ E, w ∈ Sched};
    END;
    RETURN (σ);
END;

FUNCTION SatisfyPrecedenceConstraint (v, Sched, σ, s)
BEGIN
    RETURN (max_{u ∈ Sched} σ(u)+d(u,v)−s·p(u,v))
END

FUNCTION SatisfyResourceConstraints (v, Sched, σ, s, Lb, Ub)
BEGIN
    FOR i := Lb TO Ub DO
        IF ∀0≤j<s, ρ̄_v^s(j−i mod s)+ Σ_{u ∈ Sched} ρ̄_u^s(i+j−σ(u) mod s) ≤ R
            THEN RETURN (i);
    RETURN (⊥);
END;
```

Figure 5-8: A software pipelining algorithm for acyclic graphs

5.3.3. Scheduling a cyclic graph

Here we first present the overview of the algorithm, the rationale and full details of the algorithms are described below. We preprocess the graph by finding the strongly connected components [54] in the graph, and solving the all-points longest path problem for each component [11, 25]. As the weight on the precedence constraints is expressed as a linear function of the initiation interval, the longest path algorithm is modified to operate on the symbolic value of the target initiation interval to avoid recomputing the longest paths for each initiation interval.

As in the case of acyclic graphs, the main scheduling step is iterative. For each target initiation interval, the strongly connected components are first scheduled individually. The original graph is then reduced by representing each strongly connected component as a single vertex. The resulting graph is acyclic, and the acyclic graph scheduling algorithm is used.

The scheduling algorithm for strongly connected components also uses the framework of list scheduling. The nodes are scheduled according to a topological ordering of *intra-iteration* precedence constraints. As each node is scheduled, we use the precomputed longest path information to update the range within which each remaining node must be scheduled. A node is scheduled in the earliest possible slot within the allowable range. This scheme takes advantage of the relaxation of the scheduling constraints as the initiation interval increases, as explained below.

5.3.3.1. Combining strongly connected components

In the discussion to follow, we shall use the simple cyclic graph in Figure 5-9(a) as an example. (Note that the edge from node *b* to *c* has a negative delay; negative delays arise, for example, when step *i* of a node depends on step *j* of another node, and $j<i$.) Although each machine resource is used only twice in each iteration of the loop, the lower bound of the initiation interval is three clock ticks, as determined by the cycle of edges $a \rightarrow b \rightarrow c \rightarrow a$. The nodes *a*, *b* and *c*, being nodes of a strongly connected component, are scheduled together. These nodes are then reduced to a single node *e*, as shown in Figure 5-9(b). The resulting graph is acyclic and the acyclic scheduling technique described in Figure 5-8 is applied.

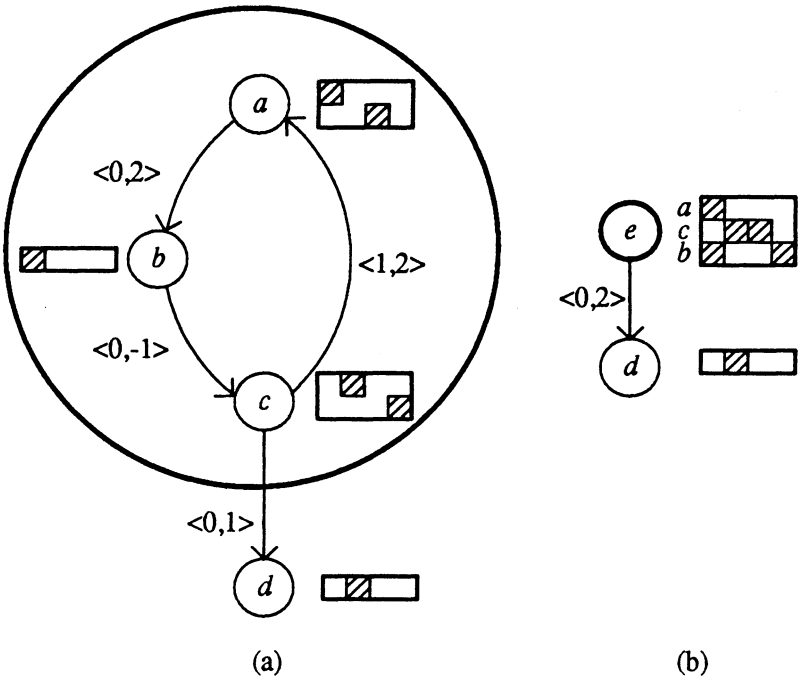


Figure 5-9: Scheduling a cyclic graph:
(a) original, and (b) reduced flow graph

In reducing the nodes in a strongly connected component into one

node, we are combining the corresponding micro-operation sequences into one long micro-operation sequence. The ith step of a sequence scheduled to initiate in clock tick n is executed in step $n+i$ of the assembled sequence; resource and precedence constraints involving step i in the sequence must therefore be transferred to step $n+i$ of the assembled sequence. In the above example, the resource usage of node e reflects the schedule and the resource usage of nodes a, b and c. The precedence constraint between nodes c and d is mapped onto an edge between nodes e and d in the reduced graph.

The formal definition of a reduced graph is given below:

Let the nodes V in graph $G=(V,E)$ be partitioned into subsets V_i, where $\cup_i V_i=V$. Suppose the nodes in the subsets V_i have been scheduled with schedules $\{\sigma_{V_i}\}$, $G'=(\{u_i\},E')$ is a reduced graph, if and only if

1. $\rho_{u_i}(j)=\sum_{v\in V_i}\rho_v(j-\sigma_{V_i}(v))$.

2. For every edge

   $(v_1,v_2)\in E$, where $v_1\in V_i, v_2\in V_j$ and $V_i\neq V_j$

   there is an edge

   $(u_i,u_j)\in E'$, where $p(u_i,u_j)=p(v_1,v_2)$
   and $d(u_i,u_j)=d(v_1,v_2)-\sigma_{V_i}(v_1)+\sigma_{V_j}(v_2)$.

   If multiple edges are defined between the same pair of nodes, then the maximum cost is used.

3. There are no more edges in E' other than those obtained from 2.

A reduced graph has the characteristic that some of the nodes in the graph may have rather heavy resource usage. In scheduling basic blocks, the resource usage is not used in prioritizing between ready nodes because it is insignificant compared to the long delays separating the nodes.

Here, nodes representing more than one node in the original graph are given a higher priority because of their greater resource requirements. The scheduling algorithm is otherwise the same.

5.3.3.2. Scheduling a strongly connected component

The same basic framework of list scheduling is used for scheduling a strongly connected component for a given initiation interval. The heuristics, however, must be revised. We cannot sort the nodes topologically; we cannot satisfy precedence constraints by merely considering those nodes that are already scheduled; and we cannot use the maximum height of the node as its priority. List scheduling is a non-backtracking algorithm and only one schedule is examined for each value of the initiation interval. We must ensure that the schedule examined has a reasonable chance of success. Experimentation with different schemes helped identify two desirable properties for the heuristics:

1. Partial schedules constructed at each point of the scheduling process should not violate any of the precedence constraints in the graph. In other words, were there no resource conflicts with the remaining nodes, each partial schedule should be a partial solution. Whenever such a partial schedule cannot be found, the process for the particular initiation interval should be considered unsuccessful and be terminated to reduce the compilation time.

2. The heuristics must be sensitive to the value of the initiation interval. An increased initiation interval value relaxes the scheduling constraints, and the scheduling algorithm must take advantage of this opportunity. It would be futile if the scheduling algorithm simply retries the same schedule that failed.

Both properties are exhibited by the heuristics in the scheduling algorithm for acyclic graphs, as well as the heuristics for strongly connected components, described below.

Satisfying precedence constraints. By the definition of strongly connected components, the entire set of precedence constraints in the component must be taken into consideration in scheduling each node. Our solution is to first precompute the maximum constraints between every pair of nodes by computing the closure of the constraints. This expensive computation is performed only once by representing the initiation interval with a symbolic value. After scheduling each node, we use this precomputed information to compute the *precedence constrained range* of the remaining unscheduled nodes. The precedence constrained range of a node for a given partial schedule is the legal range of clock ticks in which the node can be initiated without violating the precedence constraints of the graph. To ensure that all partial schedules satisfy all precedence constraints in the graph, a node is only allowed to be scheduled within its precedence constrained range.

The computation of the closure of precedence constraints is equivalent to the all-points longest path problem. Since all cycles in the graph have nonpositive costs, our all-points longest path problem is analogous to the all-points shortest path problem with no negative cycles. Dantzig et al. [11] showed that this problem can be solved using Floyd's algorithm [25].

In the closure computation, the cost of a path or an edge is represented as a function of the initiation interval s: $d(e)-s\cdot p(e)$. The precomputed result can be used for different values of s by simply substituting s with the particular initiation interval value. The cost of the longest path between two nodes is represented as a set of tuples $\{<p_1,d_1>, \ldots ,<p_n,d_n>\}$ to capture all possible maximum costs between a pair of nodes for all values of s.

As we compute the longest path between every two points, we also find all the cycles in the graph. As discussed in Section 5.3.1, the lengths of the cycles bound the value of the initiation interval s. This bound can in turn be used to reduce the size of the maximum cost set of a path. Suppose we have a set {<0,4>,<1,6>}. Since

$$\max(<0,4>,<1,6>)=\begin{cases}4, & s\geq 2\\ 6-s, & \text{otherwise,}\end{cases}$$

if we know that s is greater than 2, we can simply drop <1,6> out of the cost set.

In general, for a pair of costs $<p_1,d_1>$ and $<p_2,d_2>$, and a bound on the initiation interval, S, $<p_1,d_1>$ can be dropped from the cost set if

$p_1>=p_2$ and $d_1-d_2<(p_1-p_2)\cdot S$

The algorithm for finding the longest path and a lower bound on the initiation interval is given in Figure 5-10.

Figure 5-11 shows the closure computed using the algorithm described above for the strongly connected component shown in Figure 5-9.

Once the closure is computed, the precedence constrained range of a node can be calculated as follows: Let $Lb(u)$ and $Ub(u)$ be the current lower and upper bounds of the precedence constrained range of node u, respectively. Suppose node v is scheduled, and $<p_{vu},d_{vu}>$ and $<p_{uv},d_{uv}>$ are the maximum constraints from v to u, and v to u, respectively. Then,

$Lb(u)=\max(Lb(u),\sigma(v)+d_{vu}-s\cdot p_{vu})$, and
$Ub(u)=\min(Ub(u),\sigma(v)-d_{uv}+s\cdot p_{uv})$.

Figure 5-12(a) shows the precedence constrained ranges for nodes b and c after node a has been scheduled in time 0. The edges used in calculating the precedence constrained ranges are shown in the diagram. After scheduling node a, we have the choice to schedule b or c next. Figures 5-12(b) and (c) show the update of the remaining node for the different alternatives.

Ordering the nodes. The order in which the nodes are scheduled is crucial to the success of the algorithm. The important point is that the ordering must permit the scheduler to take advantage of the relaxed constraints as the target initiation interval is increased. Let us illustrate this

```
FUNCTION LongestPath (V, E, S)
BEGIN
    n := || V ||;
    FOR i := 1 TO n DO
        FOR j := 1 TO n DO
            C_ij := ∅;

    FOR (v_i, v_j) ∈ E DO
        C_ij := {<p(v_i,v_j), d(v_i,v_j)>};

    FOR k := 1 TO n DO
        FOR i := 1 TO n DO
            FOR j := 1 TO n DO
            BEGIN
                C_ij := MaxCost(C_ij, AddCost(C_ik, C_kj), S)

                S := max (S, max_{<p,d> ∈ AddCost(C_ji,C_ij)} ⌈d/p⌉);

            END;
    RETURN(<C, S>);
END;

FUNCTION AddCost (C_1, C_2)
BEGIN
    RETURN({<p_1+p_2, d_1+d_2> | <p_1, d_1> ∈ C_1 and <p_2, d_2> ∈ C_2});
END;

FUNCTION MaxCost (C_1, C_2, S)
BEGIN
    RETURN({<p, d> | <p, d> ∈ C_1 ∪ C_2 and ∄ <p_1, d_1> ∈ C_1 ∪ C_2,
                    where p >= p_1 and d − d_1 < (p − p_1)· S});
END;
```

Figure 5-10: Algorithm to find the longest paths and cycles in a strongly connected component

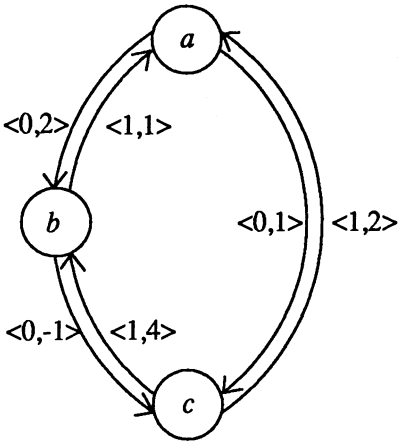


Figure 5-11: Closure of the strongly connected component in Figure 5-9

with an example. Suppose we use the deadline, the upper bound of the precedence constrained range, to prioritize the nodes. The node with the closest deadline is scheduled first. Given the graph in Figure 5-12(a), node *c* would be scheduled after node *a* because *c* has a closer deadline than *b*. Suppose there is no resource conflict between node *a* and *c*, node *c* is scheduled to execute in time 1, as early as the precedence constraints permit. Unfortunately, scheduling *c* at time 1 limits the schedulable range of node *b* to exactly one time slot, time 2. This value remains unchanged regardless of the value of the initiation interval *s*. Thus if scheduling node *b* in clock 2 happens to cause a resource conflict, it is futile to reiterate the scheduling process with a greater value of *s*.

On the other hand, if *b* is scheduled next instead of *c*, this problem does not exist. Suppose the node *b* is scheduled in time 2, the schedulable range of *c* will then be between time 1 and *s*–1. This range increases with the value of *s*. Therefore, if we cannot schedule node *c* for a given initiation interval, the chance of success improves with a larger value of *s* in each reiteration of the scheduling process.

Thus we propose the following algorithm: we schedule nodes in a topological ordering of the *intra-iteration subgraph.* The vertex set of

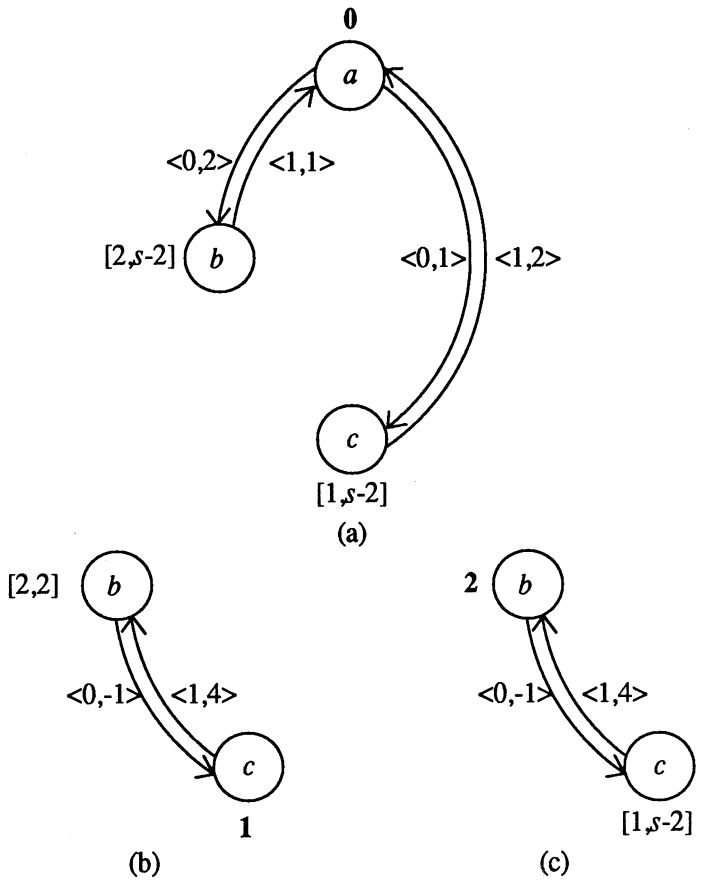


Figure 5-12: Examples of precedence constrained ranges

the intra-iteration subgraph is the same as the original graph, but the edge set contains only those edges with a zero minimum iteration difference, $p=0$. As these edges only relate operations within the same iteration, the intra-iteration subgraph must be acyclic. With this ordering, the *upper* bound on the schedulable ranges of all unscheduled nodes is always a function of the initiation interval. As the initiation interval increases, so does the chance of success. The scheduling problem approaches that of an acyclic graph as the value of the initiation interval increases. The

deadline of the precedence constrained range is only used to prioritize the ready nodes.

The scheduling algorithm of a connected component is given in Figure 5-13. These heuristics have been implemented and tested on a

```
FUNCTION SchedConnectedComponent (V, E, s)
BEGIN
    Ready := {v ∈ V | for all u ∈ V,  p(u,v) ≠ 0}
    Sched := ∅;
    FOR v ∈ V DO
    BEGIN
        Lb(v) := max_{u∈ V} d(u,v) − s·p(u,v);
        Ub(v) := ∞;
    END;
    WHILE Ready ≠ ∅ DO
    BEGIN
        v := u ∈ Ready, and Ub(u) ≤ Ub(x) for all x ∈ Ready;
        σ(v) :=SatisfyResourceConstraints
                (v, Sched, σ, s, Lb(v), min(Lb(v)+s−1, Ub(v)));
        IF σ(v) = ⊥ THEN RETURN(⊥);
        Sched := Sched ∪ {v};
        Ready := Ready−{v}+
                {u | u ∉ Sched∧∀(w,u) ∈ E ∧p(w,u)=0,w ∈ Sched};
        FOR u ∈ V − Sched DO
        BEGIN
            Lb(u) := max(Lb(u), σ(v) + d(v,u) − s·p(v,u));
            Ub(u) := min(Ub(u), σ(v) − d(u,v) + s·p(u,v));
        END;
    END;
    RETURN(σ);
END;
```

Figure 5-13: Scheduling a strongly connected component

large set of programs; the results are reported in Chapter 7. There may be many other heuristics that can produce similar results. The important

point, however, is to make sure that the scheduling heuristics have the two properties present in our algorithms: partial schedules constructed during the scheduling process should satisfy all precedence constraints in the graph, and the scheduling order must take advantage of the relaxation in constraints offered by increased values of the initiation interval.

5.3.3.3. Complete algorithm

Figure 5-14 contains the general procedure for software pipelining.

```
FUNCTION SoftwarePipeline (V, E)
BEGIN
    S := max_k ⌈ Σ_{v∈V, 0≤i<l(v)} ρ_v(i,k) / R(k) ⌉ ;
    S_max := computation time of a list schedule of the body of the loop;

    {(V_i, E_i)} := { connected components of (V, E)};

    FOR each (V_i, E_i) DO
        <E_i, S> := LongestPath (V_i, E_i, S);

    FOR s := S TO S_max DO
    BEGIN
        IF ∀ (V_i, E_i), SchedConnectedComponent (V_i, E_i, s) ≠ ⊥ THEN
        BEGIN
            (V', E') := reduced graph of (V, E);
            σ := ListSchedule (V', E', s);
            IF σ ≠ ⊥ THEN RETURN (<σ, s>);
        END;
    END;
    RETURN (<⊥, ⊥>);
END;
```

Figure 5-14: Software pipelining

We first compute a lower bound on the initiation interval as imposed by

resource constraints, and an upper bound by list scheduling one iteration of the loop. We next find the strongly connected components of the graph and compute the closure of the precedence constraints. We are then ready to find the initiation interval and schedule for the loop. We iterate within the bounds of the initiation interval; for each interval, we first try to schedule the strongly connected components individually, then schedule the reduced graph derived from the individual schedules. If any of the scheduling steps fails, we repeat the scheduling process with a greater initiation interval. Acyclic graphs can be handled by the same algorithm as each node is simply a trivial strongly connected component.

5.4. Modulo variable expansion

Modulo variable expansion is an optimization to remove unnecessary precedence constraints due to the reuse of a scalar variable in different iterations. The basic idea can be illustrated by the following code fragment in the body of a loop. A value is written into a register and used two clocks later:

```
Def(R1)
op
Use(R1)
```

If the same register is used by all iterations, then the write operation of an iteration cannot execute before the read operation in the preceding iteration. Therefore, the optimal throughput is limited to one iteration every two clocks. This code can be sped up by using different registers in alternating iterations:

```
   Def(R1),
   op,       Def(R2).
L:Use(R1),op,       Def(R1).
            Use(R2),op,       Def(R2),CJump L.
                    Use(R1),op.
                            Use(R2).
```

We call this optimization of allocating multiple registers to a variable in the loop *modulo variable expansion.* This optimization is a varia-

tion of the variable expansion technique used in vectorizing compilers [33]. The variable expansion transformation identifies those variables that are redefined at the beginning of every iteration of a loop, and expands the variable into a higher dimension variable, so that each iteration can refer to a different location. (Results of partial expressions also fall into the category of scalar variables that can be expanded). Consequently, the use of the variable in different iterations is thus independent, and the loop can be vectorized. Modulo variable expansion takes advantage of the flexibility of VLIW machines in scalar computation, and reduces the number of locations allocated to a variable by reusing the same location in non-overlapping iterations. The small set of values can even reside in register files, cutting down on both the memory traffic and the latency of the computation.

Without modulo variable expansion, the length of the steady state of a pipelined loop is simply the initiation interval. When modulo variable expansion is applied, code sequences for consecutive iterations differ in the registers used, thus lengthening the steady state. If there are n repeating code sequences, the steady state needs to be *unrolled* n times.

The algorithm of modulo variable expansion is as follows. First, we identify those variables that are redefined at the beginning of every iteration. Next, we pretend that every iteration of the loop has a dedicated register location for each qualified variable, and remove all *inter-iteration* precedence constraints between operations on these variables. Scheduling then proceeds as normal. The resulting schedule is then used to determine the actual number of registers that must be allocated to each variable. The lifetime of a register variable is defined as the duration between the first assignment into the variable and its last use. If the lifetime of a variable is l, and an iteration is initiated every s clocks, then at least $\lceil \frac{l}{s} \rceil$ number of values must be kept alive concurrently, in that many locations.

If each variable v_i is allocated its minimum number of locations, q_i,

the degree of unrolling is given by the lowest common multiple of $\{q_i\}$. Even for small values of q_i, the least common multiple can be quite large and can lead to an intolerable increase in code size. The code size can be reduced by trading off register space. We observe that the minimum degree of unrolling, u, to implement the same schedule is simply $\max_i q_i$. This minimum degree of unrolling can be achieved by setting the number of registers allocated to variable v_i to be the smallest factor of u that is no smaller than q_i, i.e.,

$$\min n, \text{ where } n \geq q_i \text{ and } u \bmod n = 0.$$

The increase in register usage is much more tolerable than the increase in code size of the first scheme for a machine like Warp.

Since we cannot determine the number of registers allocated to each variable until all uses of registers have been scheduled, we cannot determine if the register requirement of a partial schedule can be satisfied. Moreover, once given a schedule, it is very difficult to reduce its register requirement. Indivisible micro-operation sequences make it hard to insert code in a software pipelined loop to spill excess register data into memory.

In practice, we can assume that the target machine has a large number of registers; otherwise, the resulting data memory bottleneck would render the use of any global compaction techniques meaningless. The Warp machine has two 31-word register files for the floating-point units, and one 64-word register for the ALU. Empirical results reported in Chapter 7 show that they are large enough for almost all the user programs developed. Register shortage is a problem for a small fraction of the programs; however, these programs invariably have loops that contain a large number of independent operations per iteration. In other words, these programs are amenable to other simpler scheduling techniques that only exploit parallelism within an iteration. Thus, when register allocation becomes a problem, software pipelining is not as crucial. The best approach is therefore to use software pipelining aggres-

sively, by assuming that there are enough registers. When we run out of registers, we then resort to simple techniques that serialize the execution of loop iterations. Simpler scheduling techniques are more amenable to register spilling techniques.

5.5. Code size requirement

Software pipelining produces highly efficient code without greatly increasing the object code size. Figure 5-15 shows the code of a loop whose steady state has been unrolled twice. The boxes of bold lines depict segments of code containing operation sequences from different iterations. The middle section of the block in the center represents the steady state. Iterations alternatively use the "a" and "b" sets of registers, and are labeled as such. The small circle marks the instruction that decides if the end of the loop is reached, and the dashed lines represent possible branches in the flow of control.

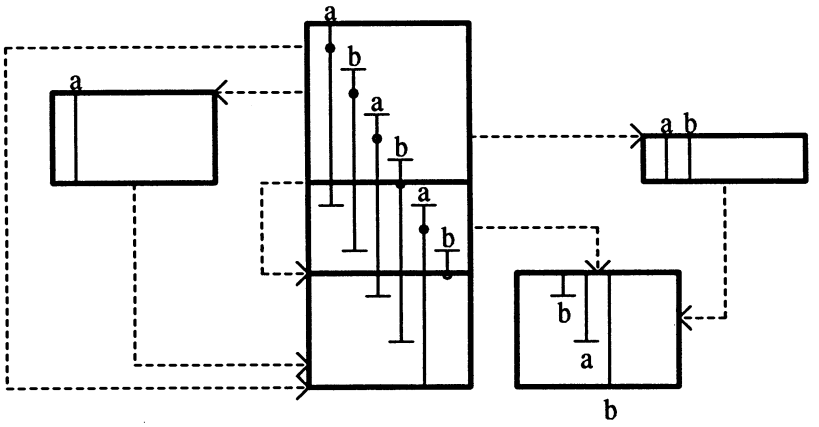


Figure 5-15: Code size requirement of software pipelining

The code length of a software pipelined loop is determined by the length of the schedule of an iteration (l), the length of code to evaluate the loop termination condition (c), the initiation interval (s) and the degree of unrolling (u). It also depends on the information available on the number of iterations in the loop. There are three cases:

1. If the number of iterations is known at compile time, we only have to generate one prolog, one steady state, and one epilog as shown in the center of the figure. The length of the loop is:

$$c+\left(\left\lceil\frac{l-c}{s}\right\rceil-1\right)s+us+(l-c-s) < l+us+l-s < 3l,$$

since $u \le \left\lceil\frac{l}{s}\right\rceil$.

2. If the loop is unrolled and the loop bound is not a compile-time constant, then multiple exits out of the steady state are needed, and we need $u-1$ more epilogs. This increases the length of the code by $(u-1)(l-c-s)$ instructions; the total length is therefore:

$$c+\left(\left\lceil\frac{l-c}{s}\right\rceil-1\right)s+us+u(l-c-s) < l+us+u(l-s) < (u+1)l$$
$$< \left(\left\lceil\frac{l}{s}\right\rceil+1\right)l.$$

3. In general, it is possible that the steady state may not even be reached; in this case, we need to generate additional prologs. The number of instructions that need to be inserted is:

$$s+2s+\ldots+\left(\left\lceil\frac{l-c}{s}\right\rceil-2\right)s < \frac{1}{2}\left(\frac{l}{s}-1\right)l.$$

Therefore, the total program length

$$< \left(u+\frac{l}{2s}+\frac{1}{2}\right)l < \left(\frac{3}{2}\left\lceil\frac{l}{s}\right\rceil+\frac{1}{2}\right)l.$$

The value $\left\lceil\frac{l}{s}\right\rceil$ is the number of iterations executed in parallel in the steady state, or the *degree of pipelining* in the software pipelined loop. The degree of software pipelining is limited to the number of concurrent operations that can be executed by the processor. The degree of pipelining is a close approximation to the speed up achieved by the software pipelining technique. Therefore the increase in code size is proportional

to the speed up obtained. This code size increase is even lower if more information is available on the number of iterations. Especially in the case of compile-time loop bounds, the code size is at most three times that of one single iteration.

Moreover, the degree of pipelining tends to be inversely related to the number of operations in the loop. It is less important to find parallelism across multiple iterations when a single iteration contains a large number of operations. This is ideal because when the degree of pipelining is high, the length of an iteration tends to be short, and we can better afford a greater factor of increase in code size.

A more important metric than the code size for the entire loop is perhaps the length of the steady state. A program is typically much smaller than the data processed, therefore the overall size of the program is not important. However, for machines with instruction caches, it is important that the innermost loop fits entirely in the cache. In other words, the size of the steady state is crucial, but not the length of the prolog and epilog. Since the steady state is only *us* instructions long, the loop is typically much shorter than the code sequence for an unpipelined iteration. Also, with the global scheduling technique described in the next chapter, the prolog and epilog can be overlapped with operations outside the loop.

If the target machine architecture cannot support the increase in code size called for in the above scheme, FOR loops can be implemented with the following alternate scheme. We handle those cases that make multiple prologs and epilogs necessary by a separate loop. The compiler generates two loops, one software pipelined and the other not. The pipelined loop contains only one prolog, a steady state, and one epilog, and the other loop is produced by simply compacting one iteration of the loop body. Let n be the number of iterations to be executed, and k be the number of iterations started in the prolog of the pipelined loop. If $n<k$, then only the unpipelined loop is executed for n iterations. Otherwise,

the unpipelined loop is executed for $n-k \bmod u$ iterations and the remaining iterations are executed on the pipelined loop. In this way, only one prolog and one epilog are needed, and the code length is at most four times the code for one iteration. The disadvantage, however, is that the prolog and epilog cannot be executed concurrently with other operations outside the loop.

5.6. Comparison with previous work

The following compares our software pipelining technique with that used in the compilers for the ESL polycyclic machine [47] and the FPS-164 computer [56].

5.6.1. The FPS compiler

Like Warp, the FPS-164 does not contain any specialized hardware to support software pipelining, and the scheduling algorithm relies on software heuristics. The algorithm for scheduling acyclic graphs presented here is similar to the algorithm used in the FPS compiler. The only difference is that the space of the initiation interval is searched sequentially in ours, whereas binary search is used in the FPS compiler. As discussed earlier, sequential search is used because schedulability is not monotonic with the initiation interval; moreover, empirical results show that a schedule can often be found with an initiation interval close to the lower bound. The major difference between the two approaches lies in the manner the cyclic graphs are handled. Also, we use modulo variable expansion to increase parallelism in the code.

The FPS compiler only software pipelines loops consisting of a single Fortran statement. Therefore, at most one inter-iteration data dependency relationship can be present in the flow graph. Although the scheduling algorithm for cyclic graphs used in the FPS compiler suffices for this restricted set of loops, it is unlikely to be effective for general cyclic graphs.

The FPS compiler handles cyclic graphs by introducing the concept of *EarlyStartTime* into the acyclic scheduling algorithm. The *EarlyStartTime* of a node can be preset before a scheduling attempt to prevent a node from being scheduled too early. Nodes are scheduled in a topological ordering of the intra-iteration subgraph. Since only precedence constraints with nodes that have already been scheduled are considered when scheduling each node, partial schedules constructed during the scheduling process do not necessarily satisfy all precedence constraints in the graph. If a node is discovered not to be schedulable because its precedence constraints are violated by the partial schedule, the culprit node is identified and its *EarlyStartTime* is modified to prevent it from being scheduled too early in the next attempt. This scheduling process is then repeated for the *same* initiation interval. While the algorithm suffices for graphs with a single inter-iteration data dependency relationship, this approach may require many attempts per initiation interval in the case of a general cyclic graph.

In our algorithm, the closure of precedence constraints is precomputed once before the first attempt of scheduling. This closure is used to update the precedence constrained range of each node cheaply in the scheduling process. The partial schedules always satisfy the precedence constraints of the graph and only one scheduling attempt is necessary for each initiation interval.

5.6.2. The polycyclic machine

The compiler for the polycyclic architecture relies on specialized hardware support. All functional units of a polycyclic machine are interconnected through a crossbar. This crossbar has storage at every crosspoint to serve as a dedicated buffer for each pair of functional units. Therefore, there is never any contention in reading or writing data by any pair of functional units. Each node in the flow graph thus consumes only one explicitly scheduled resource. This is significant because the

scheduling problem for acyclic graphs now becomes tractable [44, 47]. The minimum initiation interval can be predetermined, and an optimal schedule can easily be found. However, the problem remains NP-complete for cyclic graphs. To obtain an optimal schedule for cyclic graphs, Hsu suggested that exhaustive searching be performed on all the nodes belonging to nontrivial strongly connected components of the graph [31].

The data storage at each crosspoint has an unusual accessing scheme. It is partly a queue, and partly a register file: data can be read from any location in the storage; however, a write operation always writes to the end of the queue. In this way, addresses need to be specified only for read operations. There are two flavors of read operations; one reads the data and leaves the queue intact, and the other reads the data and deletes it from the queue. When the latter is specified, the queue is automatically compacted by moving all the data after the empty space forward by one location. This design implements modulo variable expansion without having to unroll the object code. Multiple instances of a variable are stored in the queue; they move up the queue as the queue is compacted and all instances of the variable occupy the same location by the time they are used.

The major argument for using the polycyclic crossbar is that optimal results can be obtained for acyclic loops. However, empirical results reported in Chapter 7 indicate that our scheduling heuristics can produce near-optimal schedules without incurring a heavy penalty in compilation time. In particular, most of the acyclic loops can be scheduled optimally in the first attempt in the scheduling algorithm. Moreover, with the polycyclic crossbar, exhaustive search on the nodes of all nontrivial connected components is still necessary to produce the optimal schedule. As we shall show in the next chapter, software pipelining can be extended to loops containing conditional statements. As software pipelining is applied across basic blocks, the size of the problem can become quite large and exhaustive search is unacceptable.

The second advantage of the polycyclic interconnect is that loop unrolling is not necessary. However, the analysis in Section 5.5 shows that the code size increase is tolerable. Considering the complexity of the interconnect and the width of the instruction word necessary to control the part, unrolling may be more favorable.

While the storage in the crossbar is designed to avoid unrolling, it does not have the full functionality of a register file. It is not possible to write to any random location in the queue, so it is difficult to implement global register allocation such as storing a variable in a register for the duration of an entire loop. For example, we may want to change the value of a variable within a conditional statement. Therefore, we need to write to a specific location, and not just the last location of a queue.

The major drawback of this approach is the cost of the interconnect. This interconnect requires the design of a custom VLSI component. Moreover, the data storage at every crosspoint of the crossbar requires numerous control and addressing signals; these signals imply an extremely wide micro-instruction word and thus a large micro-store. Given that our empirical results show that software heuristics can be used to generate near optimal code, the use of this specialized hardware does not seem to be justified.

5.7. Chapter summary

Software pipelining is an attractive solution to scheduling loops for highly parallel and pipelined data paths because it offers a possibility of achieving optimality with highly compact object codes. There are two intrinsic difficulties in using the technique. First, minimizing the initiation interval is an NP-complete problem; the scheduling constraints are defined in terms of the initiation interval, but the feasibility of an initiation interval is best shown by a schedule. Second, the flow graph may contain cycles due to inter-iteration data dependencies. The cycles in the graph make it difficult to design efficient non-backtracking heuristics.

These two difficulties have probably discouraged many practitioners from using this technique.

This chapter described the algorithm for scheduling both acyclic and cyclic graphs and discussed the use and implementation of the modulo variable expansion optimization. The results are that near-optimal, and often times optimal, schedules are obtained using the heuristics. They show that software pipelining is a feasible and effective scheduling technique for parallel and pipelined data paths. Good performance can be obtained by software scheduling heuristics without the need for expensive hardware.

6
Hierarchical Reduction

Hierarchical reduction is a unified approach to scheduling both within and across basic blocks. The motivation for the technique is to make software pipelining applicable to all innermost loops, including those containing conditional statements. Software pipelining has previously been defined only for loops containing straight-line code bodies. A simple conditional statement in the loop would have rendered the technique inapplicable. A loop such as the following would have been extremely inefficient on a heavily pipelined and parallel data path.

```
FOR i := 0 TO n DO
BEGIN
   IF a[i] > t THEN
   BEGIN
      count := count + 1;
   END;
END;
```

While the primary performance improvement comes from software pipelining innermost loops, there are many other global code motions

that are also desirable. For example, compacting the code within an iteration of an innermost loop is important if it consists of a large number of basic blocks and operations. Or, for innermost loops with few iterations, overlapping the prolog and epilog with scalar operations outside the loop can minimize the penalty of short vectors.

Hierarchical reduction supports all these various global code motions by a uniform mechanism. In this approach, we schedule the program hierarchically, starting with the innermost control constructs. As each construct is scheduled, the entire construct is reduced to a simple node representing all the scheduling constraints of its components with other constructs. This node can then be scheduled just like a simple node within the surrounding control construct. By capturing all scheduling constraints between the nodes, the whole range of local and global code motions is available to the scheduler. The scheduling process is complete when the entire program is reduced to a single node.

Various global scheduling, or microcode compaction techniques, have been proposed in the past. The hierarchical reduction technique is derived from the scheduling scheme previously proposed by Wood [60]. In Wood's approach, scheduled constructs are modeled as black boxes taking unit time. Operations outside the construct can move around it but cannot execute concurrently with it. Here, the resource utilization and precedence constraints of the reduced construct are visible, permitting it to be scheduled in parallel with other operations. This is essential if we want to software pipeline loops with conditional statements effectively.

Dasgupta [12] and Tokoro et al. [55] suggested global compaction be performed as a separate phase after the basic blocks have been individually compacted locally. The locally compacted code is matched against a precompiled menu of code motions, and legal code motions are performed whenever advantageous. As Fisher argues, if each block is compacted separately, too many arbitrary decisions are made during a

block's compaction that adversely affect the ability to move operations from block to block [19]. Moreover, each time an operation is moved, it opens up the possibility of more code motion. This technique is more suitable for traditional horizontal microcode engines that have much less parallelism than a high-performance processor like the Warp cell.

Fisher's trace scheduling [19] is another global scheduling technique in which the focus is on constraints of code motion. In trace scheduling, all operations from all basic blocks along the most frequently executed trace are compacted as if they belong to one basic block. All branches are handled the same way; while trace scheduling is tuned for loop branches, it does not handle conditionals well.

A technique commonly used to exploit the potential of data paths with highly pipelined functional units is vector processing. Our approach of directly translating applications into microcode bypasses the intermediate abstractions of vector instructions. Since the microcode generated by the scheduling techniques here are competitive with hand optimized code, this method greatly increases flexibility without losing efficiency.

The organization of this chapter is as follows. We first discuss how iterative and conditional constructs are handled. In the discussion, we assume that the components in the construct are all simple micro-operation sequences. We show how each type of construct, after it has been scheduled, can be represented no differently from a micro-operation sequence. Therefore, scheduling can be applied hierarchically, starting with the innermost constructs. Next, we show the various global code motions that are supported by this unified model. Finally, we conclude the chapter with a comparison with trace scheduling and the vector processing approach.

6.1. The iterative construct

Let us first describe how an iterative construct is represented after it has been software pipelined. The goal of the representation is to allow the construct be scheduled with other operations in the surrounding construct just like a straight-line micro-operation sequence inside a basic block. The representation of the node must include all necessary scheduling constraints to ensure only legal global code motions can be performed. Besides the usual resource and data dependency constraints, loops also have control constraints. The body of a loop in the object code cannot be overlapped with operations outside the loop. That is, no operations can be moved into the steady state of a software pipelined loop. The prolog and epilog, however, can be overlapped with operations outside the construct.

A software pipelined loop is modeled as a straight-line sequence of instructions, whose steps correspond to those taken when the steady state of the loop is executed exactly once. Figure 6-1 gives an intuitive picture of how the resource, data dependency and control constraints are represented in the loop node. In the figure, a dashed rectangle represents a control construct, enclosing all the operations within the construct. Each node is represented by its resource reservation table; precedence constraints are represented by arcs linking the tables.

Resource constraints. The resource requirements of the prolog and epilog can be determined by summing the resource requirements of the different instantiations of iteration in them.

Data dependency constraints. All unsatisfied precedence constraints incident on the nodes within the loop must be transferred to the iterative construct itself. To simplify the procedure, the worst case is assumed: If an operation in any of the iterations must follow some other node, we ensure that the operation in the first instantiation of the loop body obeys this constraint. Similarly, if an operation in any of the itera-

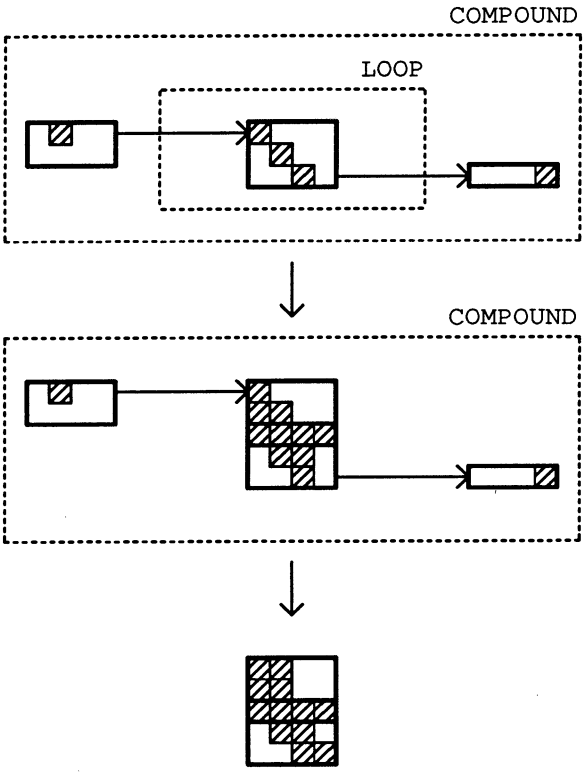


Figure 6-1: Scheduling a loop in parallel with other operations

tions must precede some other, we ensure that the operation in the last instantiation of the loop body obeys this constraint.

Control constraints. The steady state cannot be scheduled to execute in parallel with any other sequences of operations. Mutual exclusion between the iterative portion of the loop and all other operations can be expressed in the form of resource constraints. If every step of every micro-operation sequence requires at least one resource, then if the steady state segment is defined to use every resource in the system, mutual exclusion is guaranteed.

Due to the pipelining of functional units, a micro-operation se-

quence may not need a resource in every step. The initiation of a pipelined arithmetic operation may be followed by several steps where no resources are consumed. Therefore we introduce a fictitious resource into the system to implement mutual exclusion. This resource differs from all other resources in that there are infinitely many units of this resource available. Every step of every micro-operation sequence consumes one unit of this resource, so this resource does not prevent non-iterative operations from executing in parallel. Steps corresponding to the steady state, however, consume all units of this mutual exclusion resource. Since every step in a micro-operation sequence needs at least one unit of the resource, no operations are overlapped with the iterative code. (A note on the implementation: like all other resources, this resource can be modeled as a field in a bit vector. We need two bits to encode the states of consuming 0, 1 or more, and all of the resources; conflict can be detected along with other resources in a single logical operation.)

To summarize, the control flow constraints in a software pipelined loop can be disguised as resource constraints by the use of the mutual exclusion resource. This allows future scheduling steps to treat loops just like any other straight-line code sequences. General techniques, like list scheduling, can be applied. Suitable code motions, such as scheduling operations outside the loop to use the idle resources in the prolog, are automatically performed as scheduling proceeds.

6.2. The conditional construct

Because conditional statements are difficult to optimize, we concentrate on preventing conditional statements from serializing the execution of other operations. In particular, the presence of conditional statements in a loop must not prevent the loop from being software pipelined. When software pipelining a loop with conditional statements, a conditional statement may execute in parallel with operations in the same or different iterations.

Figure 6-2 illustrates the procedure for handling conditional statements. The THEN and ELSE branches of a conditional statement are first scheduled independently using list scheduling and padded to the same length. The entire conditional statement is then reduced to a single node. This node has, for its scheduling constraints, the union of the scheduling constraints of the two branches. This node can be treated within the surrounding construct like any other simple node, since any operation that can be scheduled with the union of the constraints can be scheduled with either of the two branches. At code emission time, two sets of code of equal length, corresponding to the two branches, are generated. Any code from outside the conditional statement scheduled in parallel is duplicated in both branches.

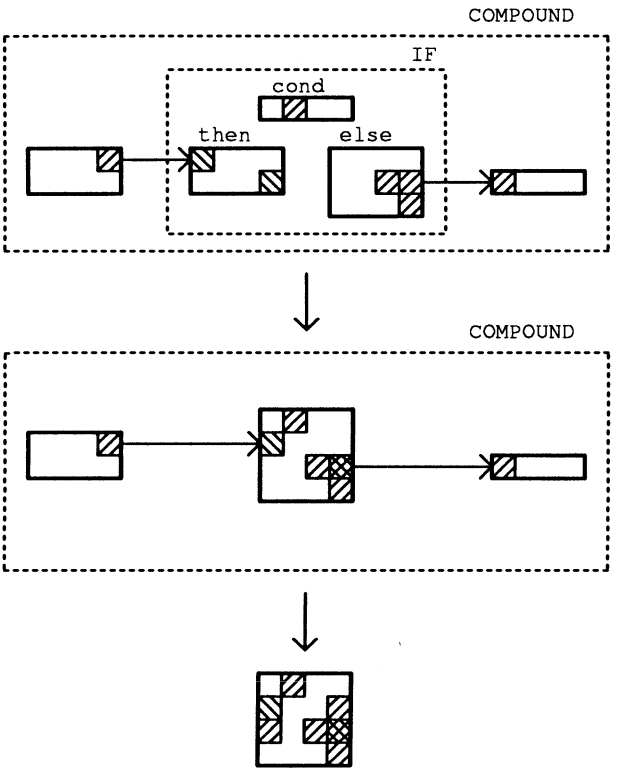


Figure 6-2: Scheduling a conditional statement

The union of the resource requirements for the two branches is computed by finding the maximum of the number of each resource used in each step. Precedence constraints between operations inside the branches and those outside must now be replaced by constraints between the node representing the entire construct and those outside. The attributes of the constraints remain the same.

If the branches of n conditional statements are overlapped together, then 2^n instructions must be generated per step to cover all the possible combinations of the outcomes of the tests in the conditionals. Compacting each branch first allows us to find the shortest schedule possible for each conditional construct; this minimizes the chances of overlapping multiple conditional statements and the increase in code size.

Representing both branches by a straight-line sequence is a simple technique that allows software pipelining to be applied to loops containing conditional statements. Although the shorter branch is extended to the same length as the longer branch, the clocks within the extension period are not completely wasted. Any resource unused by both branches in the same step can be used for operations outside the branch. Typically in a pipelined machine like the Warp processor, it is common to have many more idle resources than busy ones in a resource function of a node before software pipelining is applied.

It is difficult to predict which branch of a conditional statement is more likely to be executed, therefore strategies that optimize a branch while penalizing the other are not applicable. In our approach, the conditional statement is allowed to overlap with other operations only if the union of the scheduling constraints from both branches is satisfied, so neither branch is favored. The major advantage, however, is that the effect of conditional statements is localized; they do not disrupt the overlap of operations from other constructs and the application of the software pipelining technique.

6.2.1. Branches taking different amounts of time

The proposed strategy is designed especially to handle short conditional statements in innermost loops for machines with plenty of parallel hardware. The assumption made is that it is more profitable to satisfy the union of the scheduling constraints of both branches all the time so as not to reduce the opportunity for parallelism among operations outside the conditional statement. The assumption is not true, for example, if the long branch uses almost all the resources in its entire schedule and the other branch is very short. In particular, if we know that the long branch is taken only under exceptional circumstances, then we do not want the schedule to allow for the exceptional cases all the time. Here we discuss how we can extend the technique to allow branches to take different amounts of time.

We can control the inefficiency for having to satisfy the union of the constraints by disallowing operations inside conditional statements to overlap with those outside. Parallelism is reduced within the branches, but code motions around the construct are still allowed. This can be implemented by modifying the schedule of the branches to consume all units of the mutual exclusion resource introduced earlier. The representation of the branches is still the same as before, allowing software pipelining to be applied, if desired. When we emit the code for the branches, instructions containing no micro-operations (and not part of an indivisible sequence) are removed. By removing all resources explicitly, we avoid scheduling small numbers of operations with the branches, so the short branch remains short.

In both the original form and in this extension of the technique, the schedule for the code outside the branches is the same irrespective of the branch taken. This approach greatly simplifies the scheduling procedure.

6.2.2. Code size

The strategy of scheduling the branches of a conditional construct as compactly as possible is aimed at reducing the code size. For every conditional construct, two sets of instructions must be generated: one for each branch. When multiple conditional statements are executed concurrently, the code size increases exponentially with the degree of overlap. In software pipelining, if the length of the branches is greater than the initiation interval, then the same conditional construct from different iterations are overlapped. By compacting the branches first, we try to reduce the lengths of the branches, hence the increase in code size.

In software pipelining, scheduling branches as compactly as possible first is in conflict with the goal to minimize the initiation interval, as discussed in Chapter 5. First, it is harder to pack nodes that use large numbers of resources. Second, by packing the branches first, the schedule of a conditional node may have already violated the resource constraints for some given initiation interval. This can happen if the length of the branches is greater than the initiation interval, and that the modulo resource requirement of the node is greater than the available resources. But, as described above, this is also exactly when the code size increase can become a serious problem. Therefore, it is more important that the branch length is minimized.

Code explosion can be further controlled in other ways as well. First, side-effect free operations, such as the computation of an intermediate expression, may be moved outside the conditional statements to reduce the amount of processing within the branches. Second, we can restrict the number of overlapping conditional branches by another resource. There are n units of this "conditional construct" resource, where n is the maximum number of conditional statements that can be overlapped. After a branch of a conditional statement is scheduled, the resource requirement of the node representing the entire branch is modified by adding one unit of this special resource to the resource usage

of each step in the instruction sequence, unless it already consumes all units of the resource. The scheduler automatically enforces the limit in overlapping conditional statements in its process of avoiding resource conflicts.

6.3. Global code motions

The procedures described above allow us to represent entire control constructs as simple nodes after they have been scheduled. Once represented in such a manner, the constructs can be scheduled with other operations in the surrounding control constructs using scheduling techniques previously applicable only to basic blocks. In the W2 compiler, two scheduling algorithms are used: software pipelining is applied to all innermost loops, and list scheduling is used everywhere else. The scheduling process is performed hierarchically, starting with the innermost constructs, and is complete when the entire program is reduced to a single node.

Rearrangement of nodes representing code from different basic blocks corresponds to global code motion. By capturing all scheduling constraints between the nodes, the whole range of local and global code motions is available to the scheduler. Performing code motion at code scheduling time is obviously desirable because all relevant information to derive a good schedule is available at the same time. The following is a list of possible rearrangements of code that can take place using this approach.

1. **Rearrange a conditional statement and operations around it.** There are several possibilities:
   a. *Overlap the branches of a conditional statement with operations outside.*
   b. *Move operations around a conditional construct.*
   c. *Insert operations between the evaluation of the test and the branch operation.* In the Warp architecture, the test

in the conditional statement must be initiated many cycles before the result is available for making a branch decision. This is similar to delayed branching [29], where the branch instruction is issued but the control flow is not transferred to the branch until one or more instructions later. The cycles between the evaluation of the test and the branch instruction are treated just like other idle slots created by pipelining in the hardware. Any node that is not restricted by resource or precedence constraints can be scheduled in these slots. They may be from the same basic block, or other basic blocks. Note that operations within the branches of the conditional statement are not eligible because the branches are constrained to execute after the evaluation of the test.

d. *Insert a conditional construct within a long indivisible sequence of micro-operations.* Because of the long pipeline delay, it is possible that entire branches of a conditional statement can be embedded within some indivisible sequence of micro-operations. For example:

```
a := a+1;
if b > c then begin
   b := c;
end;
```

While the assignment of **a** takes 7 cycles on the Warp machine, the split and the rejoin of the second statement can be completed between the initiation of the addition and the assignment.

2. **Overlap multiple conditional statements.** Multiple conditional constructs unconstrained by data dependency may be overlapped just like other micro-operation sequences. An example of a program where two conditional statements may be overlapped is as follows:

```
if a > b then begin
    acount := acount + 1;
end;
if c > d then begin
    c := d;
end;
```

The second conditional statement can be executed "for free" by using the otherwise idle resources when executing the first statement. Emitting code for overlapped conditional statements is slightly more complicated. As suggested in Figure 6-3, the emitted code is not necessarily block-structured. The conditional constructs in Figure 6-3(a) are staggered by one clock unit, and the resulting code is shown in Figure 6-3(b). The arrows in the figure indicate possible control flow.

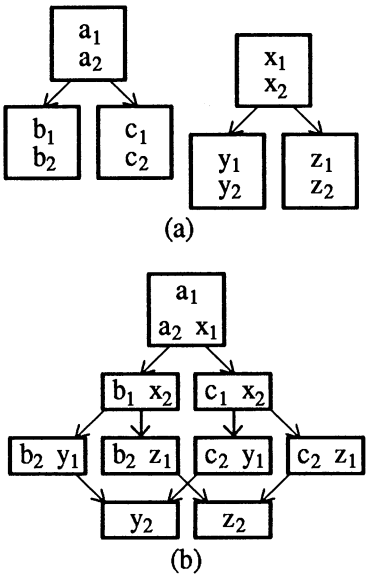


Figure 6-3: (a) Staggered conditional constructs, and (b) combined code

3. **Software pipeline a loop with conditional statements.** Since the conditional node is no different from a node representing a straight-line code sequence, software

pipelining can be applied to loops with conditional statements without modification.

4. **Overlap the prolog and epilog of a software pipelined loop with operations outside the loop.** This code motion is especially important if the number of iterations is small. It utilizes the idle resources during the filling and draining of the software pipeline, and minimizes the penalty typically associated with short vectors.

5. **Overlap the epilog with the prolog of the next loop.** This is an important special case of the above. The resource usage pattern of the epilog is generally complementary with respect to that of the prolog. As shown in Figure 6-4(a), progressively fewer resources are used in the epilog, but progressively more resources are used in the prolog. In software pipelining, filling and draining the pipelines in the functional unit are performed only once for the entire loop. By overlapping the prolog and epilog, this one time cost is shared between loops. Again, this is significant for loops with a small number of iterations.

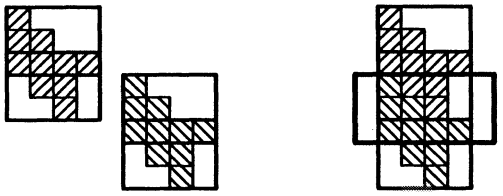


Figure 6-4: (a) An epilog overlapping with a prolog, and (b) two level software pipelining

6. **Software pipeline an outer loop.** Since a software-pipelined loop is represented no differently from any other nodes, software pipelining can be applied to outer loops as well. If the innermost loop has only a small number of iterations, the time spent executing the prolog and epilog may be significant compared to the time spent in the steady

state itself. In that case, it may be advantageous to software pipeline the outer loop as well.

The resource usages of the prolog and epilog are naturally complementary. The epilog of a loop can overlap completely with the prolog of the next instance of the loop; the resource utilization when the prolog and the epilog are overlapped is similar to that of the steady state in the original loop, as depicted in Figure 6-4(b). The steady state of the second software pipelined loop is enclosed by a box of bold lines.

6.4. Comparison with previous work

When comparing different techniques and results in global compaction, it is important to know the underlying machine models. Although the degree of pipelining and parallelism in the hardware does not affect the formulation of the global compaction problem, it determines the efficacy and suitability of the techniques used. We distinguish between two classes of machines: conventional data paths microprogrammed to emulate higher level machine architectures, and data paths consisting of large numbers of functional units for which custom microcode is generated for each application. Early global compaction work [12, 55, 60] described in the introduction of this chapter is targeted for the first category of machines; in these machines, there may be typically three or four units that can be controlled independently. The second kind of machines is epitomized by Yale's VLIW (very long instruction word) architectures [20, 24]. An example configuration of the architecture consists of 8 clusters, each containing two ALUs and two pipelined floating-point arithmetic units [15]. The Warp cell with its two 7-staged pipelined arithmetic operations and various memory storages also belongs to this class.

The fundamental difference between the microprogrammed macro architectures and VLIW class machines is that data dependency for the

latter can result in extremely sparse schedules if basic blocks are compacted locally. While rearranging operations from neighboring basic blocks is a good solution to the global compaction problem for the first kind of machines, the technique is not adequate for the second. Similarly, techniques allowing more global code motions may be unnecessarily expensive for the first. Results on experiments of global compaction must be interpreted in light of the underlying machine model.

In this section, we concentrate on the similarities and differences of our approach with other techniques intended to be used for processors with multiple parallel and/or pipelined function units. We first compare our approach to trace scheduling, then to vector processing. The vector processing approach is interesting because the characteristics of the functional units in each Warp cell are similar to those of vector machines. We will contrast our approach of custom generation of microcode from a high-level language with the use of a vector instruction set.

6.4.1. Trace scheduling

The primary idea in trace scheduling [19] is to optimize the more frequently executed traces. The procedure is as follows: first, identify the most likely execution trace, then compact the instructions in the trace as if they belong to one big basic block. The large block size means that there is plenty of opportunity to find independent activities that can be executed in parallel. The second step is to add compensation code at the entrances and exits of the trace to restore the semantics of the original program for other traces. This process is then repeated until all traces whose probabilities of execution are above some threshold are scheduled. A straightforward scheduling technique is used for the rest of the traces.

The strength of trace scheduling is that all operations for the entire trace are scheduled together, and all legal code motions are permitted. In

fact, all other forms of optimization, such as common subexpression elimination across all operations in the trace can be performed. On the other hand, major execution traces must exist for this scheduling technique to succeed. In trace scheduling, the more frequently executed traces are scheduled first. The code motions performed optimize the more frequently executed traces, at the expense of the less frequently executed ones. This may be a problem in data dependent conditional statements. Also, one major criticism of trace scheduling is the possibility of exponential code explosion [32, 39, 42, 51].

The major difference between our approach and trace scheduling is that we retain the control structure of the computation. By retaining information on the control structure of the program, we can exploit the semantics of the different control constructs, control the code motion and hence the code explosion. Another difference is that our scheduling algorithm is designed for block-structured constructs, whereas trace scheduling does not have similar restrictions. The following compares the two techniques in scheduling loop branches and conditional branches separately.

6.4.1.1. Loop branches

Trace scheduling is applied only to the body of a loop, that is, a major trace does not extend beyond the loop body boundary. To get enough parallelism in the trace, trace scheduling relies primarily on source code unrolling. At the end of each iteration in the original source is an exit out of the loop; the major trace is constructed by assuming that the exits off the loop are not taken. If the number of iterations is known at compile-time, then all but one exit off the loop are removed.

Software pipelining is more attractive than source code unrolling for two reasons. First, software pipelining offers the possibility of achieving optimal throughput. In loop unrolling, filling and draining the hardware pipelines at the beginning and the end of each iteration make optimal

performance impossible. The second reason, a practical concern, is perhaps more important. In trace scheduling, the performance almost always improves as more iterations are unrolled. Finding the suitable degree of unrolling for a particular application often requires experimentation. As the degree of unrolling increases, so do the problem size and the final code size.

In software pipelining, the *object* code is sometimes unrolled to implement modulo variable expansion, but the unrolling is performed at code emission time. Therefore, code scheduling time is unaffected. Furthermore, unlike source unrolling, there is an optimal degree of unrolling for each schedule, and can easily be determined after scheduling.

Software pipelining does not preclude source code loop unrolling. It is sometimes useful to unroll loops for reasons other than exposing parallelism to the compiler. A traditional reason to unroll loop is to reduce the overhead of loop control. If the machine instruction set does not support parallel branching, then we can reduce the cost of loop control by unrolling the loop a small number of times. (The unrolled loop body must not contain any exits out of the loop. Otherwise, a branch per iteration in the original loop is still necessary.) Loop unrolling is sometimes necessary for processors that have multiple units of identical resources. For example, if the loop body contains only a single addition and the machine has several adders, the machine can potentially execute the loop at several iterations per clock. Software pipelining can deliver at most one iteration per clock. This problem can be solved by unrolling the loop several times.

The idea of "software pipelining" has been considered in the context of trace scheduling. Fisher suggested unrolling loops as needed until a repeating pattern was observed in the object code [23]; the repeating pattern corresponds to the steady state in software pipelining. This technique is not practical because the "steady state" obtained by this approach is likely to be very large. A practical adaptation of this idea is

suggested and implemented by Su et al. [51]. Their procedure is to unroll the loop once, trace schedule the two iterations, and then reroll the loop into one. Rerolling is achieved by examining the code to locate the smallest sequence of steps that cover all the operations in one iteration. After removing redundant operations from the schedule, we obtain a steady state of the loop. This technique is a much simplified version of software pipelining, suitable only for machines with much lower degrees of concurrency. Moreover, it is unclear how this technique can be extended to handle conditional branching within a loop.

6.4.1.2. Conditionals

In the case of data dependent conditional statements, the premise that there is a most frequently executed trace is questionable. While it is easy to predict the outcome of a conditional branch at the end of an iteration in a loop, outcomes for all other branches are difficult to predict. In a study on frequencies of branches [49], branches are classified according to whether the branch target address is above or below the current instruction; almost all the branches belonging to the former class are loop branches. Results showed that for the latter class of branches, taking or not taking a branch is equally likely.

The generality of trace scheduling makes code explosion difficult to control. Some global code motions require operations scheduled in the main trace to be duplicated in the less frequently executed traces. Since basic block boundaries are not visible when compacting a trace, code motions that require large amounts of copying, and may not even be significant in reducing the execution time, may be introduced.

Ellis shows that exponential code explosion can occur by reordering conditional statements that are data independent of each other [15]. Massive loop unrolling has a tendency to increase the number of possibly data independent conditional statements. Code explosion can be controlled by inserting additional constraints between branching operations.

For example, Su et al. suggested restricting the motions of operations that are not on the critical path of the trace [51].

In our approach to conditional statements, the objective is to minimize the effect of conditional statements on parallel execution of other constructs. By modeling the conditional statement as one unit, we can software pipeline all innermost loops. The resources taken to execute the conditional statement may be as much as the sum of those of both branches. None of the operations outside the conditional statements are affected. The increase in code size due to overlapped conditional constructs can be controlled, as described in Section 6.2.2.

6.4.2. Comparison with vector instructions

Vector instructions have been a useful abstraction for managing data paths with highly pipelined functional units. The user, or a compiler, expresses the algorithm in terms of vector instructions, and the hardware manages the low level parallelism in the heavily pipelined data path. Like a vector processor, the Warp cell also has multiple highly pipelined floating-point functional units. However, it does not have a vector instruction set; microcode is directly generated from the user's application program written in a high-level language.

The alternative of microcode compilation is now made feasible by sophisticated scheduling techniques such as software pipelining. As shown in the next chapter, software pipelining can deliver performance that is comparable to hand-crafted microcode. One advantage that vector instructions have over microcode compilation is the small code size. Provided that the machine has sufficient program storage, microcode compilation offers various advantages, as described below.

In general, vector instructions increase the need for buffers to hold intermediate vectors of data. For example, a simple innermost loop of

```
FOR i := 0 TO n DO BEGIN
   d[i] := a[i]+b[i]+c[i];
END;
```

requires the computation be broken down into two vector-add instructions. If there is only one adder in the data path, chaining cannot be applied and the vector of partial sums `a[i]+b[i]` must be buffered.

Custom generation of microcode is much more flexible and adaptive to the user's program. It alleviates the need for intermediate buffering of vectors of data. For example, the microcode generated for the innermost loop above will produce a result every two cycles; the one adder in the machine is time multiplexed to implement two "vector-add" instructions. There is no need to buffer an entire vector of partial sums; as each partial sum is generated, it is routed back to the adder for the second add operation. Moreover, the latency of the operation is reduced. The first result is ready and sent to the next cell even before the last set of data arrives at the cell.

Moreover, efficient code can be generated for programs that are hard to vectorize. Software pipelining can support general recurrences in a loop, as well as conditional statements. Lastly, scalar and iterative constructs can be easily intermixed, thus significantly reducing the penalty associated with the start up of short vectors.

7
Evaluation

The ideas and techniques in this book have been validated and implemented in the W2 compilers for the Warp machines. The original W2 compiler built for the first prototype of the Warp machine was modified and retargeted as the Warp architecture evolved. The compiler was first released in late 1985 for the prototype Warp machine. Large numbers of applications including robot navigation, low-level vision, signal processing and scientific computing have been developed using this compiler by early 1987 [3, 4]. The compiler has since been retargeted to the PC production machine and iWarp, the integrated Warp architecture. While the PC machine is similar to the first prototypes, the iWarp machine is significantly different. The different compilers contain different machine-dependent optimizations but they all use the same scheduler module.

This chapter presents two sets of evaluation data: a careful analysis of performance of a large set of user programs on the first prototype Warp machine, and performance numbers for the Livermore Loops on

the PC Warp. The former analysis was performed in early 1987 before the arrival of the PC Warp. The PC machine is upward compatible with the wire-wrapped prototype. Application programs developed on the prototype run at about the same speed on the PC Warp, so we did not repeat the same analysis on the PC Warp. However, the PC Warp machine has a much greater application domain. Since the arrival of the PC Warp. many new applications have been developed and evaluated. For example, the NETtalk neural network benchmark [48] runs at 16.5 million connections per second or 70 MFLOPS on a 10 cell array [45]. Some Livermore Loop benchmark numbers on the PC Warp are presented in this chapter.

7.1. The experiment

In the analysis of user programs, a sample of 72 user programs was collected and analyzed. The overall average of the computation rate for these programs is 28 MFLOPS, that is, 28% of the peak rate of the array. We have performed experiments to study the factors governing the performance of the programs and the efficiency of the scheduling techniques. We studied the performance of entire programs, as well as the innermost loops, since they are crucial to the efficiency of a program.

Here we first present performance measurements on entire programs to provide a picture of the overall effect of the global scheduling techniques. We compare the performance of the code obtained with both a lower and an upper bound. Code generated without using any global scheduling techniques is used to provide the lower bound; comparison with this bound yields the measurement of improvement achieved by the scheduling techniques. The results are that an average speed up of three times is observed, incurring an average of 30% increase in code size. A theoretical upper bound on the performance is calculated from resource and data dependency considerations; comparing the execution times of the programs with this bound gives us the efficiency of the scheduling techniques. The results are that, on average, the efficiency is within 20% of the optimal.

We next zoom in and study the performance of software pipelining on innermost loops. We study the characteristics of the loops, measure the effectiveness of the software pipelining heuristics, and also examine various other parameters in the generated code, such as the usage of registers and the degree of unrolling. We again compare the performance of the innermost loops with a lower and an upper bound. For a lower bound, we compile the loops using only hierarchical reduction and not software pipelining; empirical results show that software pipelining improves the code by a factor of four on average, and that pipelining loops with conditional statements is feasible and is important. Comparing the performance with the upper bound obtained from resource and data dependency considerations shows that optimal throughput is achieved for at least 73% of the loops.

Before analyzing the performance data, let us first present some background material relevant to the interpretation of the data: the implementation of the compiler and the description of the programs.

7.1.1. Status of compiler

The focus in the development of the W2 compiler has been on code scheduling techniques. All the algorithms described in this book have been implemented. The W2 compiler lacks those high-level optimizations that are commonly used in vectorizing compilers. These optimizations can be applied by the user at the source level, and most programs in the sample have indeed been optimized in this manner. As these source-to-source transformations are outside the domain of the scheduler, we assume that these optimizations have already been applied when calculating the lower bound on the execution time of the program on the machine.

Loop fusion [33] is one of the optimizations commonly used in vectorizing compilers that is also applicable to this compiler. Fusing two or

more loops into one loop increases the number of independent operations scheduled together, and may lead to a faster program. For example, consider the case when one loop uses only the I/O resources and another uses only the arithmetic units. Since the I/O operations can be executed concurrently with the arithmetic operations, the execution time of the code generated for the combined loop may equal to the maximum, rather than the summation, of the execution times of the individual loops.

Another example of a useful high-level transformation is loop interchanging [58] . Since the efficiency of the innermost loops dictates the efficiency of the program, it is important that the computation of the innermost loops be able to make use of the parallel hardware. For example, the following loop

```
FOR i := 0 TO n DO BEGIN
    FOR j := 0 to n DO BEGIN
        A[i] := A[i] + B[i,j];
    END;
END;
```

sums up the ith row of **B** in **A[i]**. Since each iteration of the innermost loop needs the result of the previous iteration, different iterations cannot be overlapped. On the other hand, if the loops are interchanged as follows:

```
FOR j := 0 to n DO BEGIN
    FOR i := 0 TO n DO BEGIN
        A[i] := A[i] + B[i,j];
    END;
END;
```

The same computation is performed, but the computations in different iterations of the innermost loop are independent and can thus be overlapped.

7.1.2. The programs

The programs used in the evaluation are a sample of actual applications developed on the Warp prototype machine. Because of the restrictions on the prototype machine, these programs are subjected to the restrictions imposed by compile-time flow control. That is, they all have compile-time constant loop bounds and the conditional statements do not contain loops. The code generated for the two branches of conditional statements must execute in same amounts of time. These restrictions are no longer present in the PC Warp machine, and the compiler handles all arbitrary nesting of loops and conditional statements. All innermost loops, including WHILE loops and FOR loops of dynamic loop bounds, are software pipelined.

The programs in the experiment are applications in robot navigation, low-level vision, signal processing, and scientific computing [3, 4]. In robot navigation applications, Warp is used to process the massive amounts of low-level input data and generate high-level information for the decision making process running on the general-purpose host. The applications implemented include road following, obstacle avoidance using stereo vision, obstacle avoidance using a laser range-finder, and path planning using dynamic programming. All the W2 programs implemented for these applications are included in the analysis.

The second group of application programs comprises of routines from the SPIDER library [50], a library of functions used in vision research. There are over 100 programs from the SPIDER library implemented to date. Only programs with significantly different performance characteristics are included in the experiment.

Routines in signal processing and scientific computing included in the experiment are: singular value decomposition (SVD) for adaptive beamforming, two-dimensional image correlation using fast Fourier transform (FFT), successive over-relaxation (SOR) for the solution of

elliptic partial differential equations, routines for magnetic resonance image processing, as well as computational geometry algorithms such as convex hull and algorithms for finding the shortest paths in a graph.

A one-line description of the sample programs, their execution times and their achieved computation rates, in terms of number of floating-point operations per second, are presented in Table 7-1. Because of the restrictions imposed by the prototype machine as described above, it is possible to calculate the execution times of the programs at compile time. The computation rates are calculated by dividing the total number of floating-point operations in the program by the execution time; branches of a conditional statement are assumed to execute equal number of times.

Performance of specific algorithms on Warp		
Task All images are 512×512.	Time (ms)	MFLOPS
Mandelbrot image, 256 iterations.	6968	88.1
100×100 matrix multiplication.	25	79.4
512×512 complex FFT (per dimension).	164	71.9
3×3 convolution.	70	65.7
11×11 symmetric convolution.	367	58.8
100×100 singular value decomposition (one pass).	153	58.2
Calculate transformation table for non-linear warping.	248	57.1
Iterative enhancement of noisy image.	691	49.9
Generate matrices for plane fit for obstacle avoidance using ERIM scanner.	174	49.3
Edge detection using orthogonal templates by Frei and Chen.	301	48.4
One-iteration process for iterative enhancement of noisy images.	1415	45.5

Performance of specific algorithms on Warp		
Task All images are 512×512.	Time (ms)	MFLOPS
7×7 Average grayvalues in square neighborhood with a certain angle.	1766	44.3
Generate mapping table for affine image warping.	225	42.7
Generate diamond pattern.	277	42.4
Hough transform.	2113	42.2
SOR solving linear system of 50625 unknowns (10 iterations).	202	41.0
Edge detection using the Kirsch operator (magnitude and direction).	329	40.3
Edge preserving smoothing.	1215	39.6
Compute connectivity number.	272	39.2
Local selective averaging.	406	39.2
Generate grid pattern.	191	37.6
Generate stripe pattern.	198	36.4
Moravec's interest operator.	82	35.5
Generate bull's-eye pattern.	183	30.6
3×3 maximum filtering.	280	30.3
Sobel edge detection.	206	29.6
Convert expression modes of complex data.	697	27.8
Label color image using quadratic form for road following.	308	27.2
Iterative edge detection using Kasvand's method.	508	27.1
Obtain 0-th to second moments of two-dimensional image.	174	26.2

Performance of specific algorithms on Warp		
Task All images are 512×512.	Time (ms)	MFLOPS
Image magnification using cubic spline interpolation.	8438	25.1
Edge detection using Prewitt operator (differential type) magnitude.	206	24.5
Shrinking a binary image.	135	24.4
Growing a binary image.	135	24.4
Shortest path in 350 node graph using Warshall's algorithm (10 iterations).	104	24.3
Smooth an image while preserving texture edges.	1093	24.1
31×31 Edge value of texture edge of a specified size and direction.	395	23.5
5×5 convolution.	284	23.3
Calculate quadratic form from labeled color image.	134	22.4
Generate random numbers.	375	22.1
Measure coordinates of circumscribing rectangle.	322	21.7
Histogram over a region of an image.	185	21.5
Compute gradient using 9×9 Canny operator.	473	20.8
Discrete cosine transform on 8×8 windows.	177	20.7
3×3 Laplacian edge detection.	228	19.8
Extract or delete points in an image.	177	19.5
Compute crossing number.	225	18.9
Assign constant to inside of irregularly-shaped region.	236	18.3

Performance of specific algorithms on Warp		
Task All images are 512×512.	Time (ms)	MFLOPS
Point symmetry using analysis of variance statistical test.	47655	17.7
Inverse discrete cosine transform on 8× windows.	174	17.7
Find zero-crossings.	179	16.4
Detect borders in binary picture.	199	16.0
Roberts operator.	192	15.2
Solve matrices for plane fit in ostacle avoidance using ERIM scanner.	107	14.3
Compute the logarithm of each pixel in a real image.	468	14.0
Find next convex hull point of 1000 points.	660	13.5
Calculate (x,y,z) coordinates from ERIM laser range scanner data.	24	12.8
Generate checkboard pattern.	180	12.6
Histogram.	67	12.0
Coarse-to-fine correlation for stereo vision.	12	11.1
Scale image and set pixels outside range to constants.	147	10.9
7×7 edge value of texture edge horizontally or vertically.	143	9.3
Requantize image by reducing graylevels.	154	8.6
Measure coordinates of center of gravity.	193	8.5
Apply grayvalue translation table.	66	8.0
3×3 median filter.	448	7.8
Levialdi's binary shrink operation.	180	7.4

Performance of specific algorithms on Warp		
Task All images are 512×512.	Time (ms)	MFLOPS
31×31 average grayvalues in square neighborhood.	444	5.0
Convert real image to integer using max, min linear scaling.	249	4.4
Path planning using dynamic programming (10 iterations).	755	3.7
Image reduction by non-overlapping 2×2 windows.	154	3.0
Best-edge size, direction, and value using average grayvalues.	735	1.1

Table 7-1: Time and arithmetic computation rates of the 72 programs

The distribution of the computation rates is graphically presented in Figure 7-1. The computation rates of the programs are useful for studying the efficiency of the compiler. However, for benchmarking and comparison with other architectures, the execution times of the individual applications are more meaningful. The reason is that the number of operations executed depends on the algorithm, the compiler, and the characteristics of the machine. An algorithm efficient for an array of cells may have more arithmetic operations than that for a sequential processor. An optimizing compiler may reduce the number of operations by, for example, common subexpression elimination.

There are a few programs in the sample for which the computation rates are inflated because of the characteristics of the Warp machine. Computation within a conditional is sometimes hoisted and executed unconditionally to reduce the code size and possibly the execution time of the program, while increasing the count of floating-point operations. Because the cells on the prototype machine do not have an integer ALU, operations normally performed in fixed-point arithmetic are computed on the floating-point units and are also counted as floating-point operations.

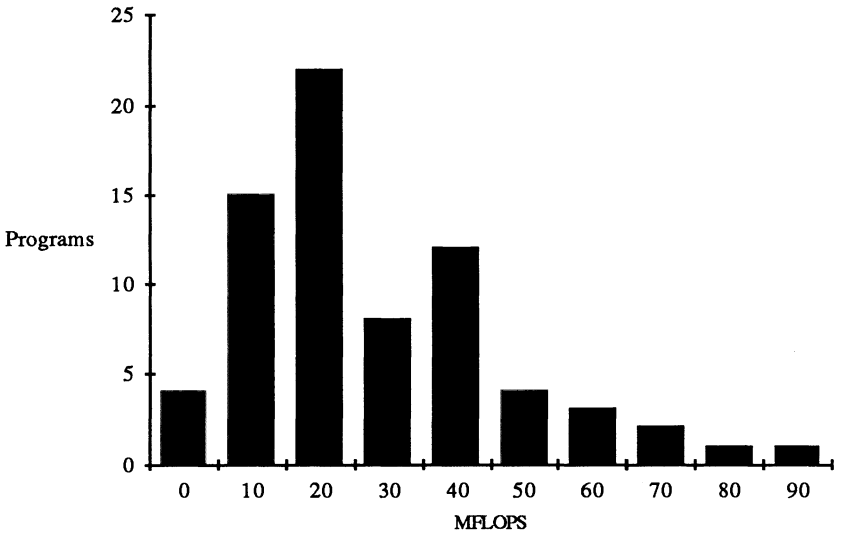


Figure 7-1: Distribution of achieved computation rates

The restrictions that the loop bounds must be compile-time constants and that conditional statements cannot contain loops also increase the arithmetic operation count. For example, efficient calculation of the Mandelbrot set requires the use of a WHILE statement. Although the MFLOPS rate of the program is high, the execution time is not as impressive because it reflects the time it takes to execute the maximum number of iterations for all pixels. We have rewritten the program for the PCWarp using WHILE constructs. The achieved computation rate is 45 MFLOPS, taking 0.2μs per iteration per pixel. Although the MFLOPS rate is lower, the program is actually much faster in all interesting cases.

All of the seventy-two programs are written for ten cells, with the only exception of the nine-cell program for 512×512 complex fast Fourier transformations. They are all homogeneous programs (all cells execute a copy of the same program), and the data flow across the array is unidirectional. The cells operate in a skewed fashion; that is, each cell starts computing shortly after its preceding cell has started. The skew

between the cells is insignificant compared to the computation time. Therefore, synchronization between cells does not impose any overhead in the computation of the array. The computation rate of the array is simply that of an individual cell multiplied by the number of cells. Therefore, in the analysis below, we can simply concentrate only on the performance of the code for each cell.

7.2. Performance analysis of global scheduling techniques

We are interested in two aspects of the global scheduling techniques: the improvement obtained from the global scheduling techniques, and how far are the scheduling techniques from the optimal.

7.2.1. Speed up of global scheduling techniques

To study the effects of software pipelining and hierarchical reduction, the scheduler is modified such that the optimizations can be applied independently. When software pipelining is used without hierarchical reduction, only innermost loops containing straight-line loop bodies are pipelined, and no other code motion is performed. When hierarchical reduction is used without software pipelining, no code motion between iterations is performed. When both global optimizations are disabled, individual basic blocks are simply locally compacted, and no code motion across any basic blocks is performed. Code produced using only local compaction is used as a baseline for comparison with globally optimized programs.

In this experiment, we compiled all the 72 programs with each combination of global scheduling techniques. Comparing the code generated using all optimizations with locally compacted code gives us a measurement of the overall improvement. The performance of the programs generated with either of the optimizations allows us to study the contribution of the individual techniques and their interaction.

Figure 7-2 shows the distribution of the factors of speed up for the various combinations of optimization techniques. The histograms plot the percentages of programs for each interval of factor of speed up. For each interval, we also show the composition of the type of programs: those that contain conditional statements and those that do not. Of the 72 programs, 42 have conditional statements in them. This classification is interesting because the significance of the scheduling techniques is different for the two groups of programs.

The global optimizations speed up the programs by a factor of 3 on average. We observe that the speed up is more significant for programs containing conditional statements. The reason is that conditional statements break up the program into even smaller basic blocks, making global scheduling techniques more important.

From the graphs in Figure 7-2, we observe that hierarchical reduction has little effect on programs that do not contain any conditional statements. Innermost loops dominate the execution time of these programs. Since the innermost loops of such programs are just single basic blocks, hierarchical reduction has no impact on the innermost loops, and therefore little effect on the overall execution time. Software pipelining is primarily responsible for the improvement in execution time.

For programs whose innermost loops contain conditional statements, hierarchical reduction can speed them up by compacting the innermost loop bodies. If hierarchical reduction is applied without software pipelining, the available parallelism is limited to that within an iteration of a loop. As shown in the figure, the improvement is noticeable but not as substantial as when software pipelining is also applied. If software pipelining alone is used, loops with conditional statements cannot be pipelined and therefore are not improved. Some speed up is still observed for programs with conditional statements because they may contain other loops that do not have conditional statements and can thus be

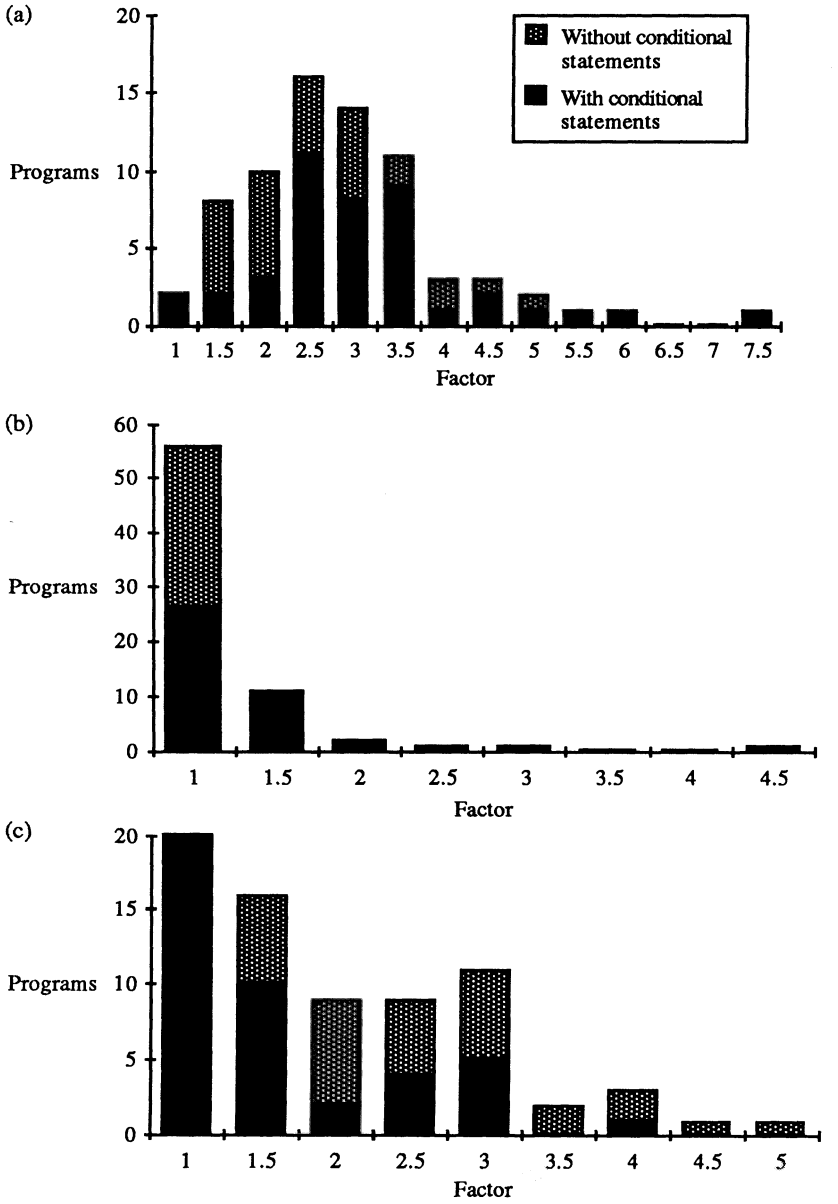


Figure 7-2: Speed up: (a) with all optimizations, (b) with hierarchical reduction only, and (c) with software pipelining only

pipelined. If both hierarchical reduction and software pipelining are applied, all innermost loops are pipelined, and a consistent improvement in performance is observed.

Figure 7-3 shows the effect of global scheduling on code size. When compared to code produced using only local compaction, an average increase by 35% is observed. Although compaction of code outside the innermost loop by hierarchical reduction typically has little effect on the execution time, it decreases the object code size by roughly 25%. Software pipelining, on the other hand, tends to increase the program size. The increase is more pronounced when software pipelining is applied to loops with conditional statements; this code expansion happens when multiple conditional statements from different iterations are overlapped.

7.2.2. Efficiency of scheduler

The peak computation rate of the Warp array is 100 MFLOPS; but, on average, the programs studied execute at 28 MFLOPS. How much of this difference between the peak rate and the achieved rate can be attributed to the inefficiency of the scheduler? We have established that global scheduling techniques speed up the programs by an average factor of 3. Here we discuss how close the achieved performance is to optimal.

The efficiency of the scheduler is measured by accounting for the execution time of the programs. There are three major components in this execution time: the exclusive I/O time, the most heavily used resource, and delay caused by data dependencies. Explanation for these components is given below. The graph in Figure 7-4 shows the breakdown of the execution times of the 72 programs.

The execution times of the programs in the figure are normalized, and the programs are sorted in increasing percentage of time accounted for by these three factors. On average, 80% of the computation time is

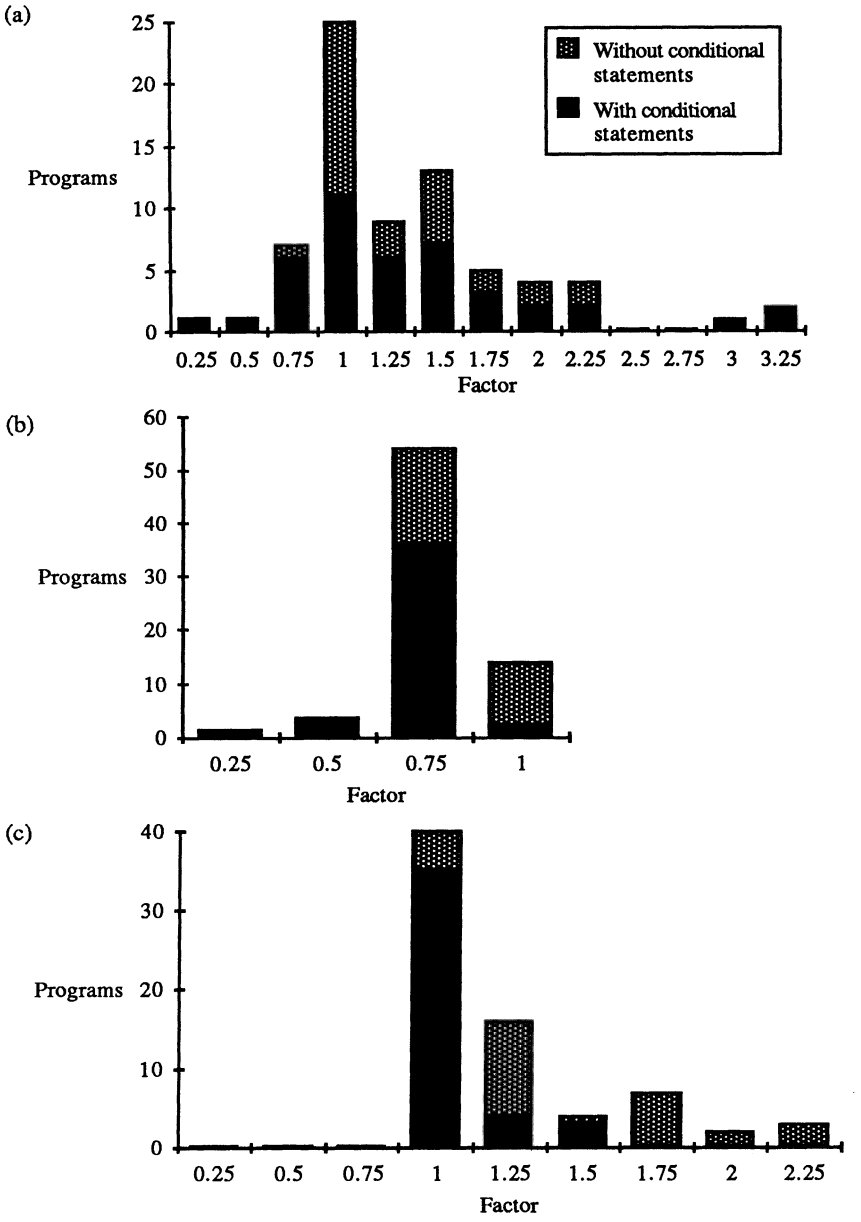


Figure 7-3: Code size increase: (a) with all optimizations, (b) with hierarchical reduction only, and (c) with software pipelining only

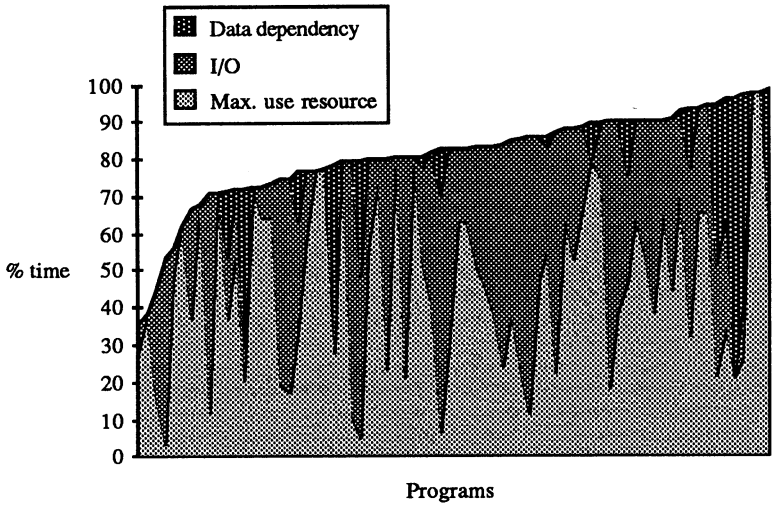


Figure 7-4: Accounting for the execution time of the programs

accounted for, therefore the scheduler's inefficiency is bounded to within the 20% of the total execution time. Figure 7-5 shows the different factors separately. The programs are again sorted in increasing time spent in each factor, and the total contribution of each factor in the execution time of all programs in the sample is given by the area below the curve.

7.2.2.1. Exclusive I/O time

One common way of using the ten cells of the Warp machine is to partition the data and computation across the array, as discussed in Chapter 3. A straightforward algorithm implementing this scheme consists of a loop with three steps: input and distribute data across cells, have each cell compute with its local data, collect and output all data to the host. In such an implementation, the input and output phases are not overlapped with the computation step. The arithmetic units are idle in the I/O phases, and vice versa. The total execution time is equal to the sum of the I/O and the computation phases.

A program with m inputs and m outputs requires $2m$ clock cycles for I/O, regardless of the number of cells in the array. The computation,

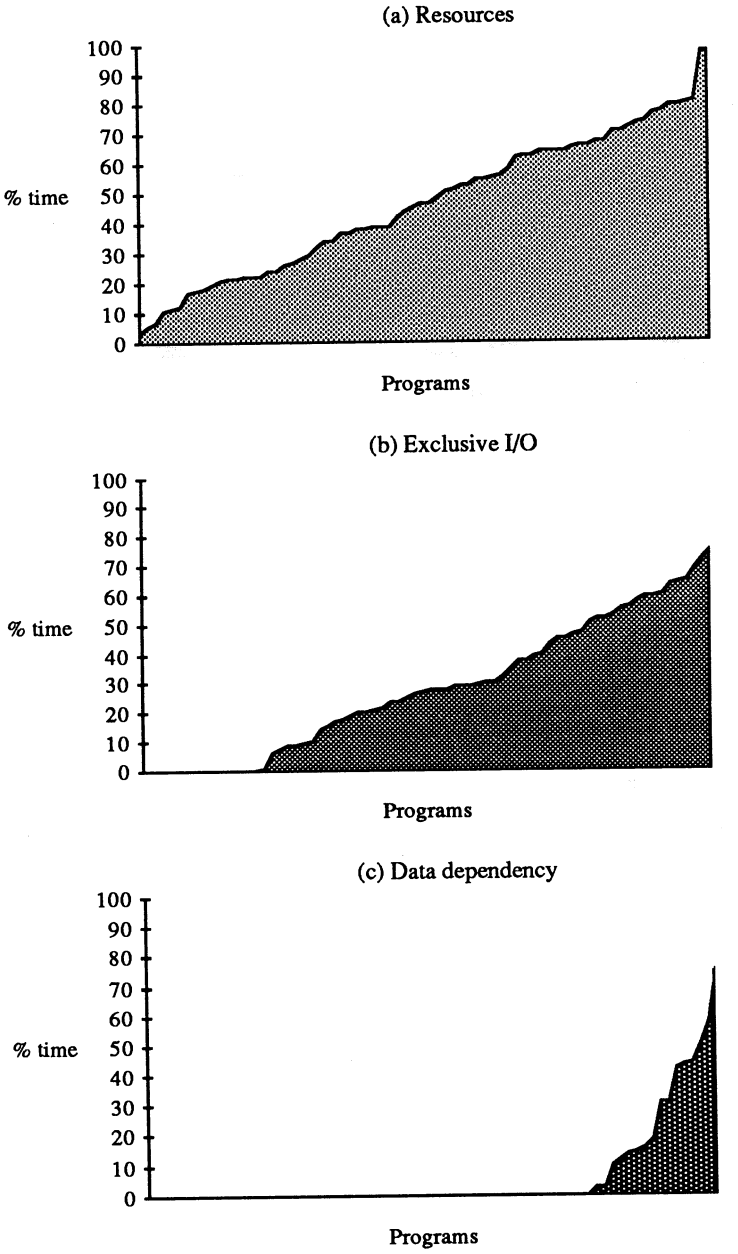


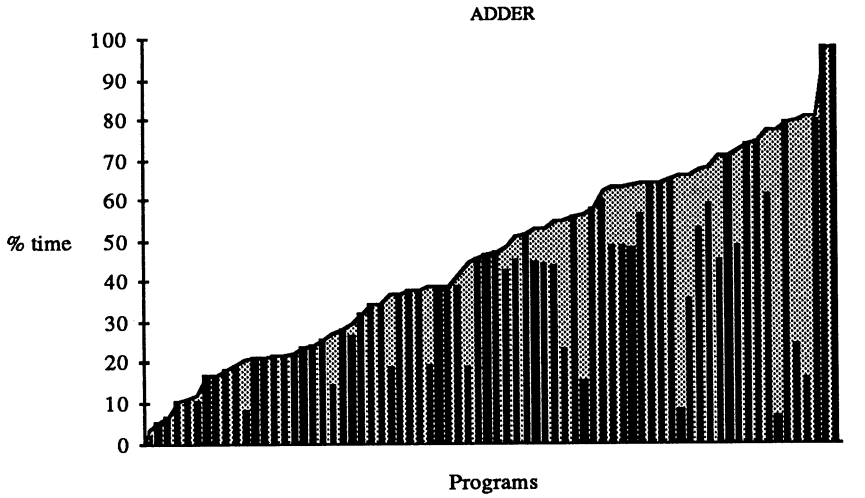
Figure 7-5: Contributions of different factors to execution time

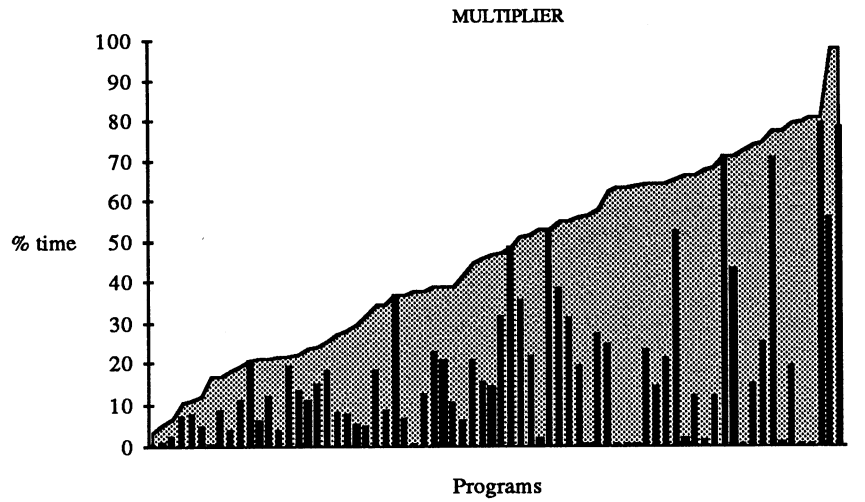
however, is divided across the n cells and takes km/n clock cycles, where k is the number of operations necessary to compute each result. In other words, while the computation time decreases with the number of cells in the system, the I/O time remains constant.

Of the 72 programs, 58 programs are written in the style described above. Many of these are vision programs operating on a fixed size (512×512) image. While a constant amount of time is spent on the I/O; the ratio of I/O to the total execution time decreases as the amount of processing increases. Figure 7-5 (b) shows that the percentage of time spent on I/O can range up to 75%. In the more conventional systolic algorithms, I/O is overlapped with the computation. Fourteen of the programs in the sample are written in this style. They do not have an *exclusive* I/O component and their execution time is the maximum of the I/O and computation time. Overlapping the I/O and computation is especially recommended when the number of operations performed per data item is small.

7.2.2.2. Global resource use count

Accounting for an average of 47% of the execution time is the use of resources. The maximum of the use counts of individual resources in the program gives a lower bound on the execution time. The use count here refers to the normalized count, obtained by dividing the true use count of the resource in the program by the number of units available in each instruction cycle. For example, unless a program has equal numbers of multiplications and additions, either the adder or the multiplier on the cell's data path must lie idle sometimes and 100 MFLOPS cannot be attained. Figure 7-5(a) shows the utilization of the most heavily used resource in each program; the use of resources in the exclusive I/O loops is not included here.


ADDER
100
90
80
70
60
% time 50
40
30
20
10
0
Programs


MULTIPLIER
100
90
80
70
60
% time 50
40
30
20
10
0
Programs

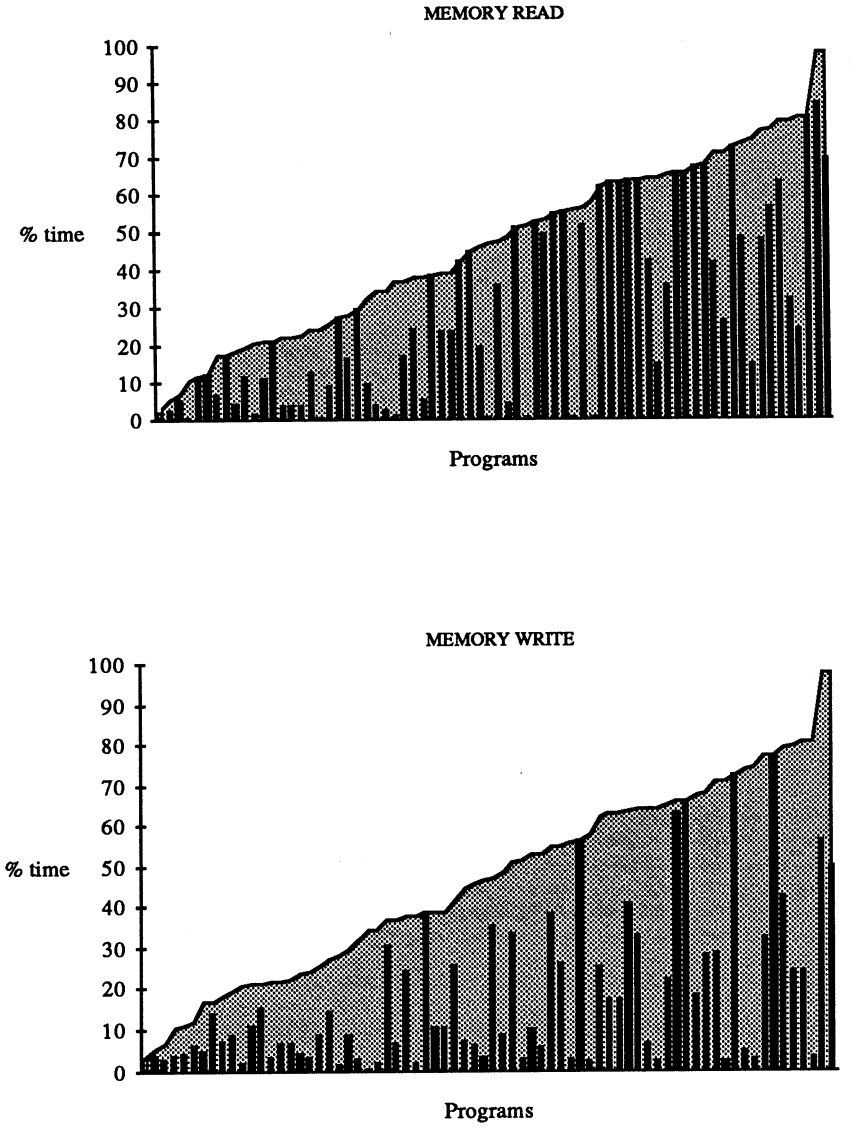


Figure 7-6: Utilization of resources in a Warp cell

There are six major resources in the prototype machine: adder, multiplier, memory reads, memory writes, X queue, and Y queue. The graphs in Figure 7-6 plot the utilization of the arithmetic units and the data memory for each of the 72 programs against a background showing the utilization of the most heavily used resource in the program. The floating-point adder is often the bottleneck, and memory read is the second most limiting resource. The queues are rarely the bottleneck, and their usages are not shown here.

7.2.2.3. Data dependency

On a pipelined and parallel machine, if a program does not have enough independent operations that can be executed concurrently, even the most heavily used resource count may not be utilized all the time.

Figure 7-5(c) shows a lower bound on the effect of data dependency on the performance of the sample programs. It plots the minimum amount of time the resource of maximum use count must be idle because of data dependency within innermost loops. From the graph, we observe that potentially only a small number of programs (16) are slowed down because of data dependency. This does not mean that the other programs do not have inter-iteration data dependencies, it is just that in some cases there are still enough independent operations in the loop to keep the resources occupied. Although the number of programs affected by data dependency is small, the effect on these programs is quite significant. Most of these programs can benefit from high level transformations such as loop interchanging, described in Section 7.1.1.

7.2.2.4. Other factors

There are several other factors contributing to the execution time that we have not accounted for. The throughput of each loop is determined by the resource that is used the most within the loop. The resource utilization shown in the Figure 7-5(a) is a global count, and thus

gives only a lower bound on the resource utilization component in the total execution time. Another factor unaccounted for is the effect of conditional control flow. The presence of conditional statements makes it impossible for any compiler to produce a finite sequence of code that is optimized for all data inputs. Therefore, the 20% of the execution time that we have not accounted for is an upper bound on the time lost due to the inefficiency of the scheduler.

7.2.3. Discussion on effectiveness of the Warp architecture

Several observations on the architecture can be made from this analysis. First, although the hardware can sustain a high peak computation rate, the effective rate is governed by the distribution of the resource use counts. For example, there are often many more additions than multiplications in a program. The Warp cell has one adder and one multiplier. Just accounting for the different numbers of additions and multiplications in the sample programs, the performance of the machine can at most be 70 MFLOPS, on average. Sometimes the memory or the I/O units are more heavily used. If all resources are considered, the achievable performance is dropped to an average of 50 MFLOPS. In other words, this value is an upper bound on the performance delivered by any compiler for the sample of programs on the machine, provided that the number of operations executed does not change. Even if there is hardware support to overlap the I/O phase with the computation of the machine, the best attainable performance is still limited to half the peak computation rate, for the sample of programs studied. This observation is, however, *not* an argument against horizontal architectures. Though the peak performance rate cannot be achieved, the obtained performance is still superior to sequential access of each resource.

Second, pipelining is an effective hardware optimization technique for implementing processors of a systolic array. Typically, applications that are suited to implementation on systolic arrays are inherently paral-

lel. The compiler techniques developed in this work allow us to find the parallelism in the code not only within basic blocks, but across basic blocks and iterations in a loop. Removing pipelining from the machine architecture does not have a dramatic effect on most of the programs in the sample; it will, however, reduce the code size of the programs.

7.3. Performance of software pipelining

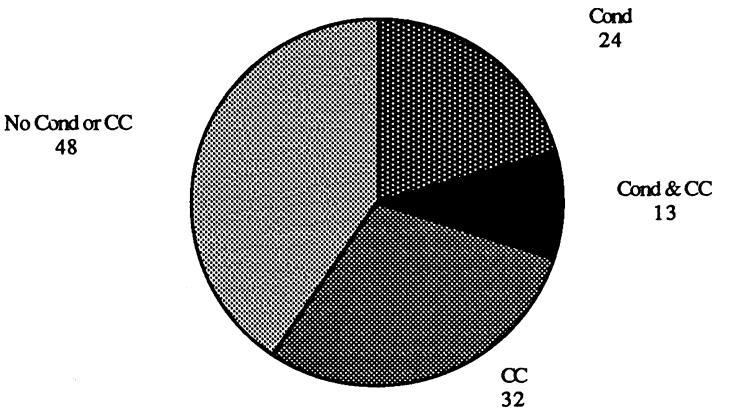
After describing the overall effects of software pipelining and hierarchical reduction on entire programs, we now further analyze the performance and the feasibility of the software pipelining algorithm by studying the individual loops in the programs. In the following, we first describe the characteristics of the loop, then the experimental results.

7.3.1. Characteristics of the loops

Only innermost loops that use the arithmetic units are included in this study. We filter out all other loops because many of the programs have identical loops where only I/O is performed. These loops are typically very simple and are invariably sped up by a factor of two or three when software pipelined. Altogether, there are 122 loops with arithmetic computation. All but five of them are pipelined successfully. In the following, we first study those that can be pipelined; we will describe the unpipelined ones at the end.

In the following performance evaluation, we distinguish among the loops according to whether they contain conditional statements, and whether they have non-trivial connected components in the flow graph. In the software pipelining algorithm, conditional constructs and non-trivial connected component are individually scheduled first before they are software pipelined with other components in the flow graph. The effect of the scheduling algorithms on these different classes of loops is different.

The breakdown of the composition of the pipelined loops in the experiment is:



There is a large variety of loops in the sample. Two representative characteristics of the loops are chosen and presented in Figure 7-7 to give a general picture of the sample: the initiation interval, and the time to execute one iteration of the source loop. The former determines the rate at which the iterations are executed. The latter quantity determines the latency in computing one iteration; it is typically only slightly longer than the execution time of one iteration if the loop is not pipelined.

7.3.2. Effectiveness of software pipelining

The throughput at which the iterations in the loop are executed is inversely proportional to the initiation interval. We can measure the effectiveness and efficiency of software pipelining by comparing the initiation intervals obtained for the loops in the sample with a lower and upper bound. For a lower bound on the initiation interval, we use the theoretical minimum initiation interval determined from the resource and precedence constraints, as described in Chapter 5. For an upper bound, we use the execution time of an iteration obtained by applying hierarchical reduction to the loop body.

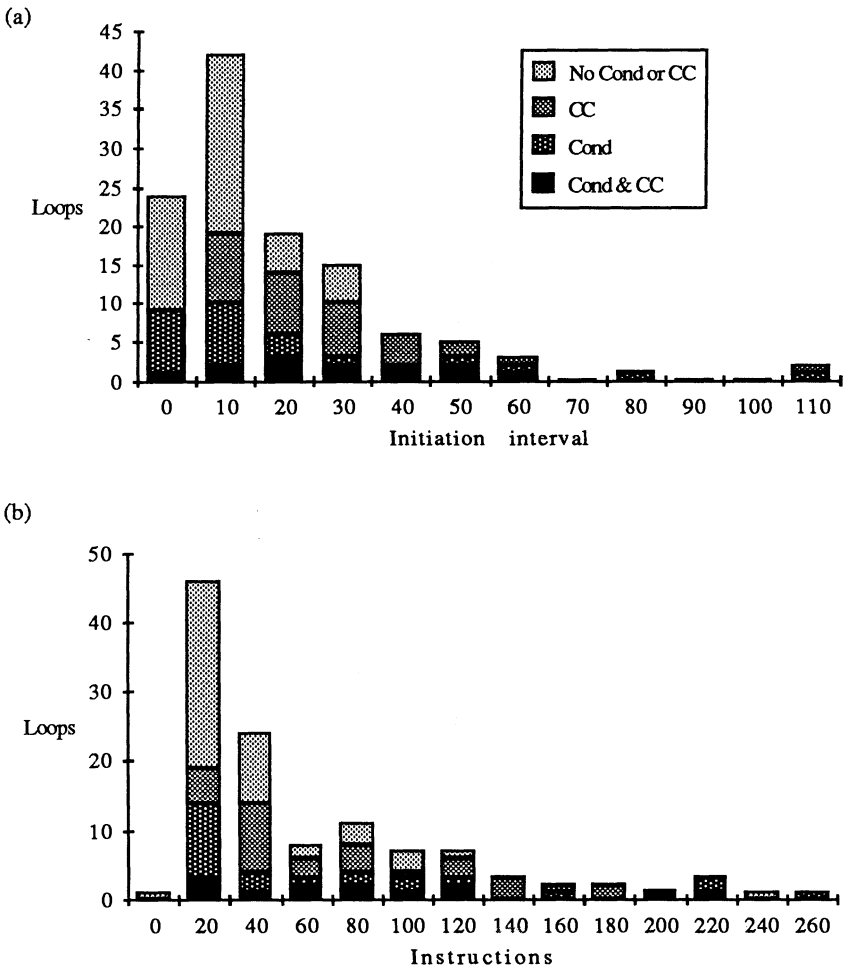


Figure 7-7: Distribution of (a) initiation intervals, and (b) execution times of single iterations

The theoretical lower bound on the initiation interval is calculated from the resource requirements and the cycle lengths in the precedence constraint graphs. In acyclic graphs, the minimum initiation interval is simply the use count of the most heavily used resource; so if a schedule meeting the minimum initiation interval can be found, then the most heavily used resource is busy all the time. In cyclic graphs, this minimum interval value may further be raised by the length of the longest

cycle in the precedence constraints graph. Cycles in the graph do not necessarily imply that there must be non-trivial connected components in the loop, because a one-node cycle would still result in a trivial connected component. On the other hand, all loops with non-trivial connected components must have cycles. About half of the loops containing non-trivial connected components have enough independent operations to theoretically keep the resources busy at all time.

For each loop, we obtain three measurements: (a) the initiation interval of the loop, (b) the execution time of a hierarchically reduced code for one iteration of the loop, and (c) the theoretical lower bound calculated as above. The potential speed up of a loop is the ratio between the execution time of an unpipelined iteration of the loop and the theoretical lower bound; the actual speed up obtained is the ratio between the execution time of an unpipelined iteration and the achieved initiation interval. The distributions of the potential and actual speed ups are shown in Figure 7-8. The two graphs show that the potential speed up is realized in most cases.

The differences between the two quantities are shown clearly in Table 7-2, which gives the number of loops meeting the lower bound, and in Figure 7-9, which shows a lower bound on the efficiency of those that do not.

Two main conclusions can be drawn from these measurements: the potential speed up of software pipelining on innermost loops is high, and that this potential is achieved in most cases. Overall, at least 85 out of 117 loops are scheduled with optimal throughput. Since we can only establish a lower bound on the initiation interval, we can measure only the lower bound of the efficiency of the algorithm on the remaining loops. As shown in Figure 7-9, the measured efficiency for these loops is rather high.

For loops with neither non-trivial connected components nor con-

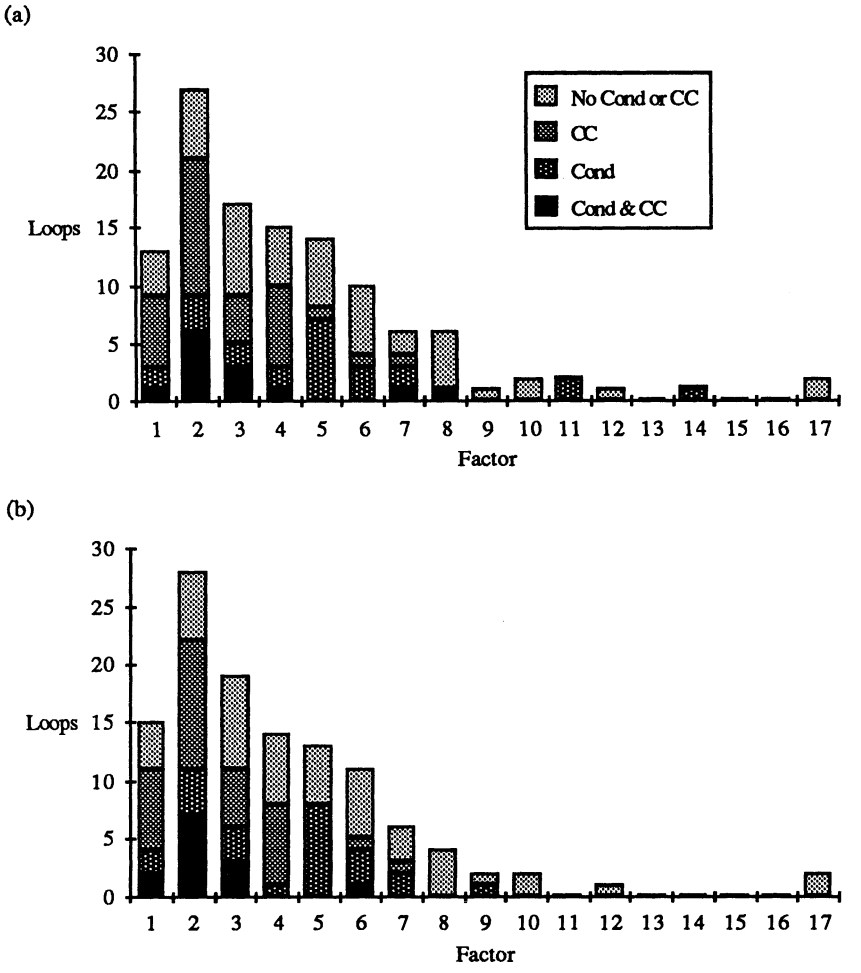


Figure 7-8: Effectiveness of software pipelining:
(a) potential and (b) achieved speed up

ditional statements, the potential speed up is spread mostly within the range from a factor of 2 to 6. Such loops are simplest to handle and the results are excellent: a large percentage of the loops meets the lower bound and the efficiency of the remaining loops is very high. The percentage of loops meeting the lower bound decreases if they contain non-trivial connected components or conditional statements. There are two reasons, as further explained below: the lower bound is not as exact and the heuristics do not perform as well.

Program	Number	Out of	%
No Cond or CC	45	48	94
CC	22	32	69
Cond	14	24	58
Cond & CC	4	13	31
Total	85	117	73

Table 7-2: Loops of 100% efficiency

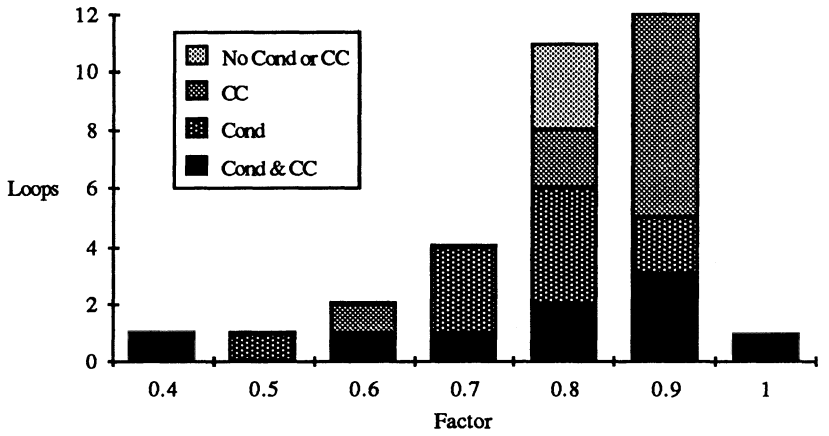


Figure 7-9: Lower bound of efficiency of remaining loops

Loops with non-trivial connected components generally have lower potential for speed up, as can be observed from Figure 7-8. The cycles in the precedence constraint graph tend to raise the initiation interval, and thus lower the potential speed up. Nodes in a connected components are more constrained in their schedule; the resulting resource conflicts are not reflected in the lower bound of the initiation interval. Also, nodes representing connected components are harder to accommodate; their scheduling constraints represent the union of their components and are therefore more stringent. These two factors lead to a lower percentage of

programs meeting the lower bound. Nonetheless, the effect of software pipelining is still significant on these loops.

Before software pipelining is applied, conditional statements are first scheduled as compactly as possible to avoid code explosion. If the branches are longer than the calculated lower bound on the initiation interval, then it is possible that the schedule of the nodes in the condition statement has already violated the resource constraints for some initiation interval values above the lower bound. Therefore, the lower bound obtained is less strict than those of other loops. The potential speed up of loops with conditional statements is similar to those without conditional statements. So, despite the slightly lower efficiency, the achieved speed up is substantial. This result shows that software pipelining loops with conditional statements is feasible and effective.

7.3.3. Feasibility of software pipelining

The software pipelining algorithm relies on several assumptions on the architecture and program characteristics. In the following, we show that these assumptions are realistic and that software pipelining is feasible.

Software pipelining iterates with increasing target initiation intervals until a schedule is found. This iterative approach is feasible because, as shown in Table 7-2, a schedule can often be found in the first attempt. Even for loops not meeting the lower bound, a schedule can typically be found within a few iterations.

In software pipelining, the object code is sometimes unrolled to implement modulo variable expansion. Object code unrolling increases the size of the object code and increases the demand of registers. Figure 7-10 shows the distribution of the degrees of unrolling in the sample loops. A degree of one means that the loop is not unrolled at all. We observe that while unrolling is used about a third of the time, the degree of unrolling, fortunately, remains small.

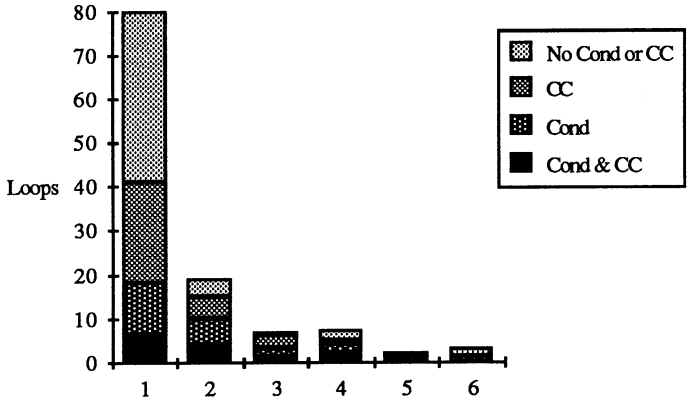


Figure 7-10: Distribution of the degrees of unrolling

In this software pipelining algorithm, register assignment is performed after the scheduling phase, and the scheduler assumes that enough registers are available. There are two 31-word register files, known as the A and M register file, in the Warp data path. They are used exclusively for storing operands for the floating-point adder and multiplier, respectively. All operands for the arithmetic units must first be stored in the register files before they can be used. Intermediate expressions are stored only in these register files, and are not written back into the data memory. In addition, registers are allocated to scalar variables in innermost loops for the duration of the loop.

The resource usage of these register files, shown in Figure 7-11, reflects the usage of the floating-point units: the utilization of the A register file is much greater than that of the M register file. In general, the assumption that there are enough registers is valid. The optimization of scheduling the graph in a top-down manner, as discussed in Section 4.3.3, reduces the lifetimes of the register values, and consequently, the register requirement. It is sometimes necessary to limit the number of registers allocated to scalar variables in the loop. And in several cases, described below, the loops cannot be software pipelined because of lack of registers.

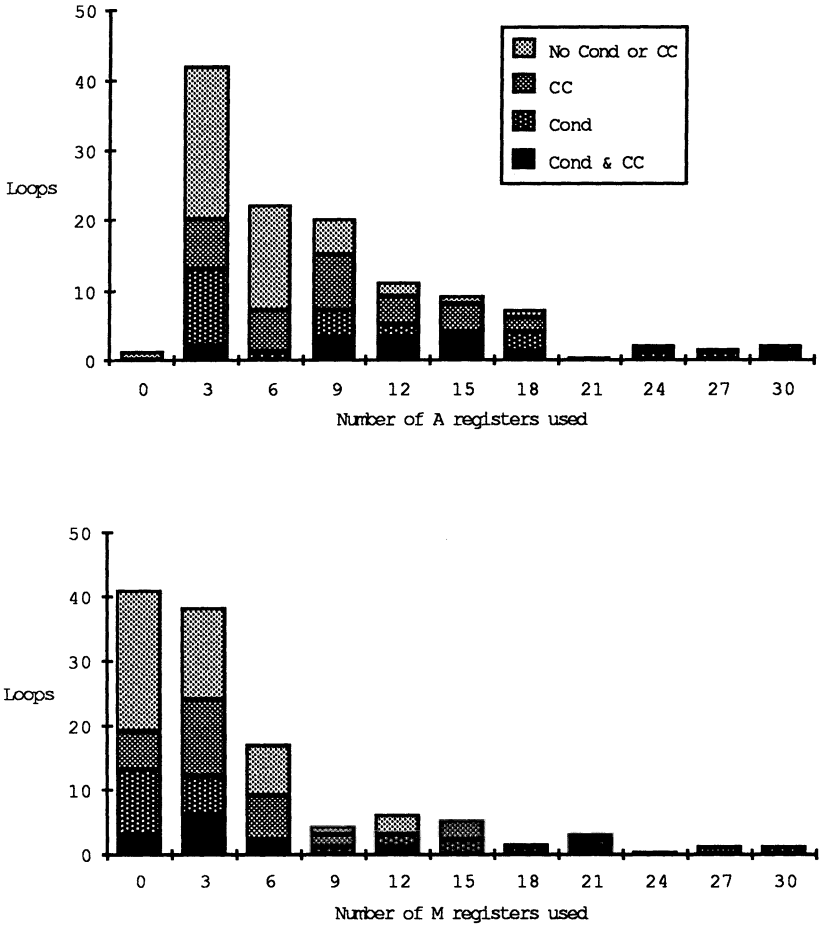


Figure 7-11: Register usage

Of the 122 loops, there are five that are not pipelined. All five loops have the same characteristics: they have unusually large loop bodies, produced by hand unrolling the loops in the source program. (Second level pipelining is preferable to source code unrolling.) Their characteristics are summarized as follows:

- Loop contains 17 connected components and 8 conditional statements.

Execution time of one iteration:	101
Minimum initiation interval:	87
Efficiency (lower bound):	86%

- Loop contains 1 connected component, which contains 60 components, 5 of which are conditional statements.

Execution time of one iteration:	142
Minimum initiation interval:	113
Efficiency (lower bound):	80%

- Loop contains 2 connected components, one of which has 13 components, 6 of which are conditional statements.

Execution time of one iteration:	72
Minimum initiation interval:	40
Efficiency (lower bound):	56%

- Loop contains 48 connected components and 8 conditional statements. A schedule is found for the initiation interval of 189 instructions, with degree of unrolling 2.

Execution time of one iteration:	207
Minimum initiation interval:	155
Efficiency (lower bound):	75%

- Loop contains 9 conditional statements and no non-trivial connected components. A schedule is found for the initiation interval of 134 instructions, with degree of unrolling 3.

Execution time of one iteration:	240
Minimum initiation interval:	132
Efficiency (lower bound):	55%

While the large numbers of operations in the loops make software pipelining difficult, they also make the technique unnecessary since the efficiency of using only hierarchical reduction is reasonably high. For the first three loops, the scheduler simply cannot find a schedule for an initiation interval shorter than the execution time of an iteration of the unpipelined loop. These results are compatible with those reported in Section 7.3.2, given that large numbers of connected components and conditional statements are present in the loops. For the last two loops, although a pipelined schedule can be found, it is not used because of the lack of registers. Both loops contain an extraordinarily large number of

objects to be scheduled, and the potential gain of software pipelining is low. In the current implementation, if the execution time of an unpipelined iteration is above a certain threshold, software pipelining is not even attempted to cut down on compilation time.

7.4. Livermore Loops

The study of the large set of user programs provides a general picture of the effectiveness of applications on the machine. However, because of the large numbers of programs used, we can only give the statistics as a whole; we cannot list the code for each program, nor can we correlate the achieved performance with the individual loop characteristics. To compensate for this, we measured the performance of a single PC Warp cell on Livermore Loops. We chose the Livermore Loops because they are standard benchmarks and performance numbers have been reported for many different compilers and machines.

The performance of Livermore Loops [43] on a single Warp cell is shown in Table 7-3. The Livermore Loops were manually translated from FORTRAN to W2. The translation into W2 was straightforward, preserving loop structures and changing only syntax, except for kernels 15 and 16, which were translated from Feo's restructured loops [16]. The INVERSE and SQRT functions expanded into 7 and 19 floating-point operations, respectively. The EXP function in kernel 22 expanded into a calculation containing 19 conditional statements. The large numbers of conditional statements made the loop not pipelinable. In fact, the scheduler did not even attempt to pipeline this loop because the length of the loop (331 instructions) was beyond the threshold that it used to decide if pipelining was feasible. Kernels 16 and 20 were also not pipelined, because the calculated lower bound on the initiation interval were within 99% of the length of the unpipelined loop.

The second column contains lower bound figures on the optimality of the software pipelining technique. As they were obtained by dividing

Kernel	Optimality (lower bound)	Speed up	MFLOPS
1	1.00	8.25	6.2
2*	0.75	3.25	2.5
3	1.00	2.71	1.4
4*	1.00	2.71	1.4
5	0.94	1.12	0.6
6*	1.00	2.86	1.4
7	1.00	6.00	7.9
8*	1.00	2.29	8.2
9	1.00	4.27	7.7
10	0.85	5.31	3.4
11	0.90	1.30	0.5
12	1.00	4.00	1.7
13	1.00	2.63	2.0
14**	1.00	3.32	2.2
15	0.85	5.50	5.9
16	1.00	1.00	0.3
17	1.00	1.20	0.8
18	0.97	3.70	7.5
19	1.00	1.24	0.7
20	0.99	1.00	0.9
21	1.00	6.00	3.0
22***	0.56	1.00	1.1
23	1.00	1.10	1.1
24	1.00	1.33	0.4
H-Mean			1.2

* Compiler directives to disambiguate array references used

** Multiple loops were merged into one

*** EXP function expanded into 19 IF statements

Table 7-3: Performance of Livermore Loops

the lower bound on the initiation interval by the achieved interval value, they represent a lower bound on the achieved efficiency. For kernels containing multiple loops, the figure given is the mean calculated by weighing each loop by its execution time. The performance of the Livermore Loops is consistent with that of the users' programs. Except for kernel 22, which has an extraordinary amount of conditional branching due to the particular EXP library function, near-optimal, and often times, optimal code is obtained. The speed up factors in the third column are the ratios of the execution time between an unpipelined and a pipelined kernel. All branches of conditional statements were assumed to be taken half the time. The MFLOPS rates given in the last column are for single-precision floating-point arithmetic.

The figures in the second column indicate that optimal efficiency is achieved for most of the Livermore Loops. Although consistently good results are obtained, the speed up fluctuates dramatically. The achieved MFLOPS rates range from 0.4 to 8.2 MFLOPS. First, we note that there is very little correlation between the optimality and the achieved execution rates. The MFLOPS rate of the kernels 3 to 6 are very low, but their optimality figures are typical and are reasonably high. On the other hand, kernels 7 to 9 have high MFLOPS rates and are 100% optimal. There is a high potential of parallelism in these loops, and the scheduler is able to exploit it fully.

In general, the performance of the generated code depends mainly on the characteristics of the program. If there is parallelism within and across iterations of a loop, the scheduler is able to find it and exploit it. As discussed in Section 7.2.2, parallelism is limited by data dependency and the distribution of resource usage:

1. ***Data dependency.*** Consider kernel 5 of the Livermore Loops:

```
FOR i := 0 TO n-1 DO BEGIN
    X[i] := Z[i]*(Y[i]-X[i-1]);
END;
```

The multiplications and additions are serialized because of the data dependencies. Even if the adder and the multiplier were not pipelined, only one unit can be busy at any one time. But since the multiplier and the adder are seven stage pipelined, each iteration takes 14 clocks. Just by this consideration alone, Warp is limited to a peak performance of 0.7 MFLOPS on this loop.

Inter-iteration data dependency, or recurrences, do not necessarily mean that the code is serialized. This is one important advantage that VLIW architectures have over vector machines. As long as there are other operations that can execute in parallel with the serial computation, a high computation rate can still be obtained.

2. ***Critical resource bottleneck.*** The execution speed of a program is limited by the most heavily used resource. Unless *both* the floating-point multiplier and adder are the most critical resources, the peak MFLOPS rate cannot be achieved. Programs containing no multiplications cannot run faster than 5 MFLOPS since the multiplier is idle all the time. For example, since the integer unit is the most heavily used resource in kernel 13, the MFLOPS measure is naturally low.

7.5. Summary and discussion

Many systolic array programs have been developed for the Warp prototype machine, despite the restrictions of compile-time flow control and homogeneous programming. This is not surprising because all previously proposed systolic algorithms in the literature belong to this restricted class of computation. All the programs in the sample have unidirectional data flow, and the compiler is able to generate code that fully utilizes the array of cells. Once a cell is activated, after possibly a small delay, it never stalls waiting for data. By rearranging I/O opera-

tions at code scheduling time, the code generated for a cell is just as efficient as it would have been if the cell did not have to interact with other cells.

At the cell level, our analysis shows that the cell code scheduler performs well for the sample of user programs in the experiment. More precisely, an average of a three-fold increase in speed has been observed, and, on average, the efficiency of the scheduler is shown to be at least 20% within optimal. Software pipelining is effective; optimal throughput is achieved for at least 73% of the loops. Hierarchical reduction is also important: It makes software pipelining applicable to loops with conditional statements, and allows consistent speed up be achieved for all programs.

Although an average of 28% utilization is a respectable figure for high-performance processors, there seems a large discrepancy between the achieved performance and the measured efficiency of the scheduler. The analysis in Section 7.2.2 provides an answer to this question. The major reason is that the high peak rate of the Warp array is achieved through parallel and pipelined functional units. As the execution time is limited by the most heavily used resource, the effective computation rate of the machine is generally much lower than the peak. For the sample of programs studied, if the critical resource in every cell is busy at all times, the average performance obtained would still only be 50 MFLOPS. This number can be improved only by increasing the bandwidth of the functional units in the data path. Modifications, such as reducing the number of pipelined stages in the arithmetic units, do not change this figure.

The analysis also pinpoints one potential area for improvement. The computation and the I/O units are not used in parallel in many of the programs where data partitioning model of computation is used. This model is extremely powerful and should be supported effectively. Overlapping the computation and I/O phases is difficult because they are two unrelated tasks with different control flow. In many cases, it is possible

to apply high-level transformations to merge the different phases in software. However, a general solution to this problem is to provide independent sequencing control for the I/O process. This support is provided in iWarp.

8

Conclusions

The Warp architecture represents a significant breakthrough in systolic computing. Systolic arrays have previously been known to be highly efficient, but special-purpose hardware architectures. The cost of dedicated hardware is so prohibitive that although many different systolic algorithms have been designed, few have been implemented. The Warp project demonstrates that programmability can be achieved without sacrificing efficiency. Even in its prototype form, the Warp machine can already deliver execution speeds for many numerical oriented applications that rival existing supercomputers. The availability of the single-chip iWarp processor will make a significant impact on the practice of parallel computing. Arrays of thousands of cells are feasible, programmable, and much cheaper than many other supercomputers of comparable power.

This book studies one of the vital aspects of the Warp machine: programmability. My thesis is that high-performance systolic arrays can be programmed effectively using a high-level language. The ideas and

techniques in the book have been validated by the W2 compiler implementation, which has been used extensively by a large user community. Optimal performance is obtained for many simple, classical systolic programs, such as convolution and matrix multiplication. The availability of the language and the compiler have made possible the development and implementation of many new, complex systolic algorithms. The compiler provided the programmability and efficiency essential to making Warp a valuable resource in computation intensive domains. This research has extended the domain of systolic processing from simple mathematical recurrences in custom VLSI implementations to arbitrarily complex programs on powerful and programmable processors.

Specifically, this work makes contributions to two major research areas: a machine abstraction and compiler optimizations for systolic arrays, and code scheduling techniques for horizontally microcoded processors.

8.1. Machine abstraction for systolic arrays

We propose to model a high-performance systolic array as an array of simple sequential processors with asynchronous communication. While this machine model is well-known in general computing, its applicability to systolic arrays is not apparent. This is because systolic algorithm designers typically exercise extreme control over the timing of the communication between cells to guarantee a high utilization of the cells. This work shows that, in fact, for the very reason of efficiency, we should select the asynchronous communication model. The internal timing of a parallel and pipelined processor interacts with the array level of parallelism, and this timing must be considered in generating efficient code for the array. Therefore, if the internal parallelism of the cells in an array were to be hidden from the user by the use of a high-level programming language, then the timing of the interaction between cells would also have to be under the jurisdiction of the compiler.

Efficiency typical of systolic array algorithms can be obtained for unidirectional systolic algorithms written under this asynchronous communication model. The high-level semantics of asynchronous communication avoids over-specification in the interaction between cells. Simple analysis can be performed independently on each cell program to relax the original ordering of communication operations in the sequential program. This allows the scheduler to rearrange the communication operations to use the internal parallelism effectively. The sequencing constraints between communication operations are modeled in a manner similar to data dependency constraints between computational operations. This correspondence of systolic array specific constraints to well known concepts in code scheduling allows us to apply established tools and algorithms to our specific problem.

Our machine abstraction hides two low-level issues: cell implementation details and synchronization between cells. This abstraction represents a significant data point in the space of machine models for systolic arrays. First, it offers both efficiency and generality. Our experience in using the model and the compiler supporting the model indicates that there is no need for lower level tools. This machine abstraction can be used as a target machine model for higher level programming tools. It is likely that different higher level tools will be developed to capture characteristics of different array usage models. Our abstraction may serve as a common base model for all these different approaches. Second, the machine abstraction can be used for a wide range of hardware implementations. Although the asynchronous communication model is used in the abstraction, it is not necessary that this communication model be supported directly in hardware. This is important because many systolic algorithms, including all previously published ones, do not need dynamic flow control. We have shown that this machine abstraction is recommended even for machines without the dynamic flow control hardware because it is amenable to compiler optimizations.

8.2. Code scheduling techniques

This research work shows that software pipelining is a *practical*, *efficient*, and *general* technique for scheduling the parallelism in a horizontally microcoded or VLIW machine. Previous implementations of software pipelining either depend on specialized hardware, or limit the kind of loops to which it can be applied. This research develops software pipelining into a complete algorithm that is based on software heuristics. In particular, the algorithm for cyclic program graphs has been significantly improved. Software pipelining of innermost loops is the primary reason for the high attainable performance on Warp.

This work also proposes a unified approach to scheduling both within and across basic blocks called hierarchical reduction. By modeling an entire construct in a manner similar to an operation in a basic block, all scheduling techniques previously applicable only within a basic block are now applicable to all constructs. The important effects on execution time are: loops with conditional statements can be pipelined, innermost loops containing large numbers of blocks and operations can be effectively compacted, the prologs and epilogs of innermost loops can be overlapped with scalar operations outside the loops. Hierarchical reduction is important in ensuring that a consistent performance improvement can be obtained for all programs.

The algorithms of software pipelining and hierarchical reduction are relatively simple, and can be implemented without undue effort. Comparison with the small sample of painstakingly hand-microcoded applications indicates that the generated code is at least as efficient. The experimental data on a large set of programs shows that optimal throughput can be obtained for the innermost loops of many of the applications running on the Warp machine.

The study of the performance of the compiler and application programs on Warp leads to several observations on architecture design.

First, the effective use of the highly parallel and pipelined data paths by compiler-generated code suggests a new alternative to vector processing. Custom generation of microcode is much more flexible and adaptive to the user's program. It alleviates the need for intermediate buffering of vectors of data. Moreover, efficient code can be generated for programs that are hard to vectorize. Software pipelining can support general recurrences in a loop, as well as conditional statements. Lastly, scalar and iterative constructs can be easily intermixed, thus significantly reducing the penalty associated with short vectors.

Second, the systolic architecture is a feasible and effective parallel machine organization. Systolic arrays possess two fundamental attractive attributes: their scalable architecture offers a solution to meet the ever-growing demands on computation speed, and their fast local interconnection between cells uniquely supports algorithms of fine-grain parallelism effectively. The Warp machine demonstrates that a programmable systolic array of high-performance processors is feasible, and this research shows that the power of a systolic array can be harnessed without undue programming efforts. We have laid a foundation for further research in both systolic architectures and higher level programming tools. With the development of more sophisticated hardware and software support, systolic arrays will develop into a major resource for computation intensive applications.

References

[1] Aho, A. V., Sethi, R., and Ullman J. D.
Compilers.
Addison-Wesley, 1986.

[2] Aiken, A. and Nicolau, A.
Perfect Pipelining: A New Loop Parallelization Technique.
Technical Report, Cornell University, Oct., 1987.

[3] Annaratone, M., Bitz, F., Clune E., Kung H. T., Maulik, P., Ribas, H., Tseng, P., and Webb, J.
Applications of Warp.
In *Proc. Compcon Spring 87*, pages 272-275. IEEE Computer Society, San Francisco, Feb., 1987.

[4] Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P. C., Tseng, P., and Webb, J. A.
Applications Experience on Warp.
In *Proc. 1987 National Computer Conference*, pages 149-158. AFIPS, Chicago, June, 1987.

[5] Brent, R. P. and Kung, H. T.
Systolic VLSI Arrays for Polynomial GCD Computation.
IEEE Transactions on Computers C-33(8):731-736, August, 1984.

[6] Chen, M.
Space-Time Algorithms: Semantics and Methodology.
Technical Report 5090:TR:83, California Institute of Technology, May, 1983.

[7] Chen, M.
A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI.
In *Proc. 13th Annual ACM Symposium on Principles of Programming Languages.* Jan., 1986.

[8] Coffman, E. G., Jr.
Computer and Job-shop Scheduling Theory.
John Wiley & Sons, 1976.

[9] Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K. .
A VLIW Architecture for a Trace Scheduling Compiler.
In *Proc. Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 180-192. Oct., 1987.

[10] Cydrome Inc.
CYDRA 5 Directed Dataflow Architecture.
1987.

[11] Dantzig, G. B., Blattner, W. O. and Rao, M. R.
All Shortest Routes from a Fixed Origin in a Graph.
In *Theory of Graphs*, pages 85-90. Rome, July, 1967.

[12] Dasgupta, S.
The Organization of Microprogram Stores.
Comput. Surv. 11(1):39-65, March, 1979.

[13] Delosme, J.-M., Ipsen, I. C. F.
Design Methodology for Systolic Arrays.
In *Proc. SPIE Symp.*, pages 245-259. 1986.

[14] Ebcioglu, K.
A Compilation Technique for Software Pipelining of Loops with Conditional Jumps.
In *Proc. 20th Annual Workshop on Microprogramming.* Dec., 1987.

[15] Ellis, J. R.
Bulldog: A Compiler for VLIW Architectures.
PhD thesis, Yale University, 1985.

[16] Feo, J. T.
An Analysis of the Computational and Parallel Complexity of the Livermore Loops.
Parallel Computing 7(2):163-186, June, 1988.

[17] Fisher, J. A.
The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources.
PhD thesis, New York Univ., Oct., 1979.

[18] Fisher, J. A.
2^n-Way Jump Microinstruction Hardware and an Effective Instruction Binding Method.
In *Proc. 13th Annual Workshop on Microprogramming*, pages 64-75. November, 1980.

[19] Fisher, J. A.
Trace Scheduling: A Technique for Global Microcode Compaction.
IEEE Transactions on Computers C-30(7):478-490, July, 1981.

[20] Fisher, J. A.
Very Long Instruction Word Architectures and the ELI-512.
In *Proc. Tenth Annual Symposium on Computer Architecture*, pages 140 - 150. Stockholm, June, 1983.

[21] Fisher, J. A., Ellis, J. R., Ruttenberg, J. C. and Nicolau, A.
Parallel Processing: A Smart Compiler and a Dumb Machine.
In *Proc. ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 37-47. Montreal, Canada, June, 1984.

[22] Fisher, A. L., Kung, H. T. and Sarocky, K.
Experience with the CMU Programmable Systolic Chip.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII*. Society of Photo-Optical Instrumentation Engineers, August, 1984.

[23] Fisher, J. A., Landskov, D. and Shriver, B. D.
Microcode Compaction: Looking Backward and Looking Forward.
In *Proc. 1981 National Computer Conference*, pages 95-102. 1981.

[24] Fisher, J. A. and O'Donnell, J. J.
VLIW Machines: Multiprocessors We Can Actually Program.
In *Proc. Compcon Spring 84*, pages 299-305. IEEE Computer Society, February, 1984.

[25] Floyd, R. W.
Algorithm 97: Shortest Path.
Comm. ACM 5(6):345, 1962.

[26] Fortes, J. A. B., Moldovan, D. I.
Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms.
Journal of Parallel and Distributed Computing 2:277-301, 1985.

[27] Garey, M. R. and Johnson, D. S.
Computers and Intractability A Guide to the Theory of NP-Completeness.
Freeman, 1979.

[28] Gross, T. and Lam, M.
Compilation for a High-performance Systolic Array.
In *Proc. ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 27-38. June, 1986.

[29] Gross, T. R. and Hennessy, J. L.
Optimizing Delayed Branches.
In *Proc. 15th Annual Workshop on Microprogramming*, pages 114-120. 1982.

[30] Gross, T., Kung, H. T., Lam, M. and Webb, J.
Warp as a Machine for Low-Level Vision.
In *Proc. IEEE International Conference on Robotics and Automation*, pages 790-800. March, 1985.

[31] Hsu, P.
Highly Concurrent Scalar Processing.
PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[32] Isoda, S., Kobayashi, Y., and Ishida, T.
Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph.
IEEE Transactions on Computers c-32(10):922-933, October, 1983.

[33] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, M.
Dependence Graphs and Compiler Optimizations.
In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207-218. January, 1981.

[34] Kung, H. T.
Why Systolic Architectures?
Computer Magazine 15(1):37-46, January, 1982.

[35] Kung, H. T.
Memory Requirements for Balanced Computer Architectures.
Journal of Complexity 1(1):147-157, 1985.

[36] Kung, H. T. and Lam, M.
Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays .
Journal of Parallel and Distributed Computing 1(1):32-63, 1984.
A preliminary version appears in Proc. Conference on Advanced Research in VLSI, MIT, January 1984, pp. 74-83.

[37] Kung, H. T. and Webb, J.A.
Global Operations on the CMU Warp Machine.
In *Proc. 1985 AIAA Computers in Aerospace V Conference*, pages 209-218. American Institute of Aeronautics and Astronautics, Oct., 1985.

[38] Kung, H. T. and Webb, J. A.
Mapping Image Processing Operations onto a Linear Systolic Machine.
Distributed Computing 1(4):246-257, 1986.

[39] Lah, J. and Atkin, E.
Tree Compaction of Microprograms.
In *Proc. 16th Annual Workshop on Microprogramming*, pages 23-33. Oct., 1982.

[40] Lam, M. and Mostow, J.
A Transformational Model of VLSI Systolic Design.
Computer 18(2), Feb., 1985.
An earlier version appears in Proc. 6th International Symposium on Computer Hardware Description Languages and their Applications, May, 1983.

[41] Leiserson, C. E. and Saxe, J. B.
Optimizing Synchronous Systems.
Journal of VLSI and Computer Systems 1(1):41-68, 1983.

[42] Linn, J. L.
SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information.
In *Proc. 16th Annual Workshop on Microprogramming*, pages 11-22. 1983.

[43] McMahon, F. H.
Lawrence Livermore National Laboratory FORTRAN Kernels: MFLOPS.
1983.

[44] Patel, J. H. and Davidson, E. S.
Improving the Throughput of a Pipeline by Insertion of Delays.
In *Proc. 3rd Annual Symposium on Computer Architecture*, pages 159-164. Jan., 1976.

[45] Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. and Kung, H. T.
Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second.
In *Proceedings of 1988 IEEE International Conference on Neural Networks*, pages 143-150. July, 1988.

[46] Quinton, P.
Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations.
In *Proc. 11th Annual Symposium on Computer Architecture*. 1984.

[47] Rau, B. R. and Glaeser, C. D.
Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing.
In *Proc. 14th Annual Workshop on Microprogramming*, pages 183-198. Oct., 1981.

[48] Sejnowski, T. J., and Rosenberg, C. R.
Parallel Networks that Learn to Pronounce English Text.
Complex Systems 1(1):145-168, 1987.

[49] Shustek, L. J.
Analysis and Performance of Computer Instruction Sets.
PhD thesis, Stanford University, May, 1977.

[50] Electrotechnical Laboratory.
SPIDER (Subroutine Package for Image Data Enhancement and Recognition).
Joint System Development Corp., Tokyo, Japan, 1983.

[51] Su, B., Ding, S. and Jin, L.
An Improvement of Trace Scheduling for Global Microcode Compaction.
In *Proc. 17th Annual Workshop in Microprogramming*, pages 78-85. Dec., 1984.

[52] Su, B., Ding, S., Wang, J. and Xia, J.
GURPR – A Method for Global Software Pipelining.
In *Proc. 20th Annual Workshop on Microprogramming*, pages 88-96. Dec., 1987.

[53] Su, B., Ding, S. and Xia, J.
URPR – An Extension of URCR for Software Pipeline.
In *Proc. 19th Annual Workshop on Microprogramming*, pages 104-108. Oct., 1986.

[54] Tarjan, R. E.
Depth first search and linear graph algorithms.
SIAM J. Computing 1(2):146-160, 1972.

[55] Tokoro, M., Tamura, E. and Takizuka, T.
Optimization of Microprograms.
IEEE Transactions on Computers c-30(7):491-504, July, 1981.

[56] Touzeau, R. F.
A Fortran Compiler for the FPS-164 Scientific Computer.
In *Proc. ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 48-57. June, 1984.

[57] Weiss, S. and Smith, J. E.
A Study of Scalar Compilation Techniques for Pipelined Supercomputers.
In *Proc. Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 105-109. Oct., 1987.

[58] Wolfe, M. J.
Optimizing Supercompilers for Supercomputers.
PhD thesis, University of Illinois at Urbana-Champaign, October, 1982.

[59] Woo, B., Lin, L. and Ware, F.
A High-Speed 32 Bit IEEE Floating-Point Chip Set for Digital Signal Processing.
In *Proc. 1984 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 16.6.1-16.6.4. 1984.

[60] Wood, G.
Global Optimization of Microprograms Through Modular Control Constructs.
In *Proc. 12th Annual Workshop in Microprogramming*, pages 1-6. 1979.

[61] Young, D.
Iterative Solution of Large Linear Systems.
Academic Press, New York, 1971.

Index