January 1986

UILU-ENG-86-2203
CSG-49

COORDINATED SCIENCE LABORATORY
College of Engineering

HIGHLY CONCURRENT SCALAR PROCESSING

Peter Yan-Tek Hsu

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS none
2a. SECURITY CLASSIFICATION AUTHORITY none	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-86-2203 (CSG 49)	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION 1. Joint Services Electronics Program 2. Naval Electronics Systems Command VHSIC Program
6c. ADDRESS (City, State and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code) JSEP: Office of Naval Research; 800 Quincy Street; Arlington, VA 22217 VHSIC: 2511 Jefferson Davis Highway; Arlington, VA 22202
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program; Naval Electronics Systems Command VHSIC Program 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER JSEP: N00014-84-C-0149 VHSIC: N00039-80-C-0556
8c. ADDRESS (City, State and ZIP Code) JSEP: Office of Naval Research; 800 Quincy St.; Arlington, VA 22217 VHSIC: 2511 Jefferson Davis Highway, Arlington, VA 22202	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: N/A PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Highly Concurrent Scalar Processing	
12. PERSONAL AUTHOR(S) Peter Yan-Tek Hsu	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) January, 1986 15. PAGE COUNT 119
16. SUPPLEMENTARY NOTATION none	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) compiling, scheduling, optimization, parallel processing, pipelined processing, horizontal architectures, guarded instructions

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

High speed scalar processing is an essential characteristic of high performance general purpose computer systems. Efficient concurrent execution of scalar code is difficult due to data dependencies and conditional branches. This thesis proposes a code scheduling heuristic called the decision tree scheduling (DTS) technique for general scalar code, and an optimal code scheduling algorithm called the simple loop scheduling (SLS) algorithm for a restricted class of innermost loops. Also proposed is a highly concurrent machine architecture that takes advantage of these scheduling techniques.

The DTS technique performs extensive code rearrangement over a complex of basic blocks to achieve high levels of speedup. This technique is based on a software implementation of well-known hardware speedup techniques for instruction pipelines, including out-of-order execution, branch prediction, and branch lookahead with conditional execution. To support the DTS technique we propose an architectural concept called guarded instructions. Guarded store instructions enhance a compiler's ability to reorder loads and stores so as to increase the level of concurrency. Guarded jump instructions allow the execution of conditional

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) 22c. OFFICE SYMBOL none

branches to be overlapped, thereby significantly reducing the average branch time. Performance evaluation of the DTS technique based on realistic but problematic workloads drawn from the UNIX kernel and other sources are presented.

The SLS algorithm exploits the regular structure of simple innermost loops to generate optimal throughput loop code. This approach is an adaptation and extension of the theory on optimal design of hardware pipelines. This algorithm is shown to be of optimal complexity, and highly efficient in practice. Related issues of register assignment and branch handling are discussed and resolved.

The proposed scheduling techniques are most useful for highly concurrent architectures; both parallelism and pipelining can be exploited efficiently. A tightly coupled heterogeneous multiprocessor with appropriate support for the DTS and SLS techniques is presented. This multiprocessor can be implemented using currently available technology, and is sufficiently flexible to accommodate both general purpose processors and specialized functional units, with an appropriate mix of parallelism and pipelining. System configurations can be adapted to resolve cost-performance tradeoffs for particular applications and technologies.

HIGHLY CONCURRENT SCALAR PROCESSING

BY

PETER YAN-TEK HSU

B.C.S., University of Minnesota, 1979
M.S., University of Illinois, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

HIGHLY CONCURRENT SCALAR PROCESSING

Peter Yan-Tek Hsu, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1986
Edward Davidson, Advisor

High speed scalar processing is an essential characteristic of high performance general purpose computer systems. Efficient concurrent execution of scalar code is difficult due to data dependencies and conditional branches. This thesis proposes a code scheduling heuristic called the *decision tree scheduling* (DTS) technique for general scalar code, and an optimal code scheduling algorithm called the *simple loop scheduling* (SLS) algorithm for a restricted class of innermost loops. Also proposed is a highly concurrent machine architecture that takes advantage of these scheduling techniques.

The DTS technique performs extensive code rearrangement over a complex of basic blocks to achieve high levels of speedup. This technique is based on a software implementation of well-known hardware speedup techniques for instruction pipelines, including out-of-order execution, branch prediction, and branch lookahead with conditional execution. To support the DTS technique we propose an architectural concept called *guarded instructions*. Guarded store instructions enhance a compiler's ability to reorder loads and stores so as to increase the level of concurrency. Guarded jump instructions allow the execution of conditional branches to be overlapped, thereby significantly reducing the average branch time. Performance evaluation of the DTS technique based on realistic but problematic workloads drawn from the UNIX kernel and other sources are presented.

The SLS algorithm exploits the regular structure of simple innermost loops to generate optimal throughput loop code. This approach is an adaptation and extension of the theory on optimal design of hardware pipelines. This algorithm is shown to be of optimal complexity, and highly efficient in practice. Related issues of register assignment and branch handling are discussed and resolved.

The proposed scheduling techniques are most useful for highly concurrent architectures; both parallelism and pipelining can be exploited efficiently. A tightly coupled heterogeneous multiprocessor with appropriate support for the DTS and SLS techniques is presented. This multiprocessor can be implemented using currently available technology, and is sufficiently flexible to accommodate both general purpose processors and specialized functional units, with an appropriate mix of parallelism and pipelining. System configurations can be adapted to resolve cost-performance tradeoffs for particular applications and technologies.

ACKNOWLEDGEMENT

I am deeply grateful to my thesis advisor, Professor Edward Davidson, for his understanding and encouragement. His insightful guidance has been essential to this work, and his friendship has made the effort most enjoyable.

I wish to express my gratitude to my previous advisors, Professor Michael Schlansker and Professor B. R. Rau, for their helpful suggestions in the initial part of this research.

I would like to thank my examination committee, Professors Campbell, Kuck, Lawrie, and Patel, for their time and helpful comments. I would also like to thank my colleagues and the secretaries in the Computer Systems Group of the Coordinated Science Laboratory for their assistance.

I am very grateful to my wife Mary Prisco for her patience and support. I also thank my parents for their encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Motivation and Research Objective

There is unquestionably a need for high-speed general-purpose computer systems: the range of applications handled by computers as well as the volume of data processed by computers is continuously increasing. Advances in circuit technology have resulted in dramatic performance improvements. However, circuit technology advances alone have proven insufficient in satisfying the increasing demand for higher performance.

By providing a high degree of concurrency through parallelism and pipelining, modern supercomputers are capable of delivering significantly higher performance for applications dominated by numerical computations. In contrast, the extensive use of concurrency to achieve higher performance for applications dominated by nonnumeric or symbolic computations has met with only limited success.

The objective of this research is to investigate new techniques that use concurrency to improve the performance of nonnumeric/symbolic computation-intensive applications. The approach taken by this research is an integrated design philosophy in which the machine organization and instruction set architecture are developed in conjunction with the development of the compiler's code generation strategy.

1.2. Overview of this Work

This work is concerned with achieving highly concurrent processing of scalar code, i.e. code without vector instructions. Concurrency is most easily obtained for code that is readily vectorizable. Inherently scalar code, i.e. code that cannot be vectorized,

causes severe performance degradation in most concurrent machines. Such code occurs to some extent in all applications and dominates in nonnumeric and symbolic applications. Inherently scalar code is often characterized by

(i) prolific use of data-dependent conditional branches with very little computation between successive branches, and

(ii) use of linked data structures with memory address pointers, as opposed to array data structures with integer indices.

We have developed compiler code generation techniques, architectural support features, and a machine organization that specifically addresses the problems of highly concurrent scalar computation.

Chapter 2 focuses on the problem of conditional branches. In this chapter we propose a code generation heuristic, called the *decision tree scheduling* (DTS) technique, that performs extensive code rearrangement over a complex of basic blocks to achieve high levels of speedup. The DTS technique is based on a software implementation of well-known hardware speedup techniques for instruction pipelines, including out-of-order execution, branch prediction, and branch lookahead with conditional execution.

To support the DTS technique we propose an architectural concept called *guarded instructions*. Guarded store instructions enhance a compiler's ability to reorder loads and stores, thus increasing the average level of concurrency. Guarded jump instructions allow the execution of conditional branches to overlap, significantly reducing the average time per transfer of control.

The DTS technique is most useful for highly concurrent architectures; both parallelism and pipelining can be exploited efficiently. We present performance evaluation of the DTS technique based on realistic but problematic workloads drawn from the UNIX kernel and other sources. This evaluation is performed by evaluating concurrency

relative to a Cray-1-like scalar unit by considering a pipelined processor with similar timing and the capability of issuing one or more instructions per clock cycle.

Chapter 3 focuses on the problem of code generation for program loops, with a running example that manipulates linked data structures. Because the traversal of linked data structures introduces special problems, conventional loop-speedup techniques such as vectorization and multitasking are shown to be unusable and/or less cost-effective. In this chapter we present a code generation algorithm, called the *simple loop scheduling* (SLS) algorithm, that generates throughput-optimal loop code for a class of loops that does not contain nested conditional statements. The significance of throughput-optimal loop code is that, while the loop is executing in steady-state, peak performance is continuously maintained.

Chapters 2 and 3 deal primarily with compiler-based code optimization techniques. These code optimization techniques were developed based on a fairly detailed machine model. Chapter 4 describes the machine model and discusses implementation considerations that motivated the particular choice of machine organization. Several machine dependent code generation issues, including the problem of register allocation, are also discussed in this chapter.

The main results of this research are summarized in chapter 5. In this chapter we also present some suggestions for future research in this area.

CHAPTER 2

SCHEDULING SCALAR CODE

2.1. Introduction

High speed scalar processing is an essential characteristic of high performance general purpose computer systems. Pipelined instruction execution is the standard method for increasing scalar performance beyond performance levels achievable by fast logic technology alone[1, 2]. Unfortunately, the potential throughput of instruction pipelines is rarely achieved because scalar code usually contains many data dependencies and conditional branches. These disrupt the smooth flow of instructions which causes the pipeline to be underutilized, leading to performance degradation.

Many techniques have been proposed for improving the throughput of instruction pipelines. Well known techniques include out-of-order execution[3], branch prediction[4], and branch lookahead (conditional issue of further instructions while awaiting branch outcomes)[2]. Although effective, implementing these techniques via hardware increases the complexity of pipeline control and usually causes the clock cycle to be lengthened as well. Thus the performance advantage gained by increasing the pipeline utilization by such techniques is degraded by both an increase in cost and a reduction in the clock speed.

In this chapter, we propose a software implementation of these techniques, with modest hardware support, that minimizes these disadvantages. This approach relies on an integrated design philosophy in which the machine architecture is developed in conjunction with the development of the compiler's code generation strategy. This idea represents an extension of similar philosophies reported in the literature for processors

with very short instruction pipelines (in the range of two to four segments)[5, 6, 7, 8, 9].

The ideas of this chapter naturally extend to much longer instruction pipelines and multiple-pipeline parallel architectures. The potential throughput of an instruction pipeline is increased by partitioning the instruction pipeline into more segments with finer granularity, thereby increasing the length of the pipeline but speeding up the clock. Delivered performance rarely approaches the potentially higher instruction issue rate of longer pipelines due to the increased difficulty of efficiently utilizing a longer pipeline. The instruction set modifications, their hardware support, and the code optimization techniques proposed here are most useful for such highly-concurrent scalar architectures.

2.2. Motivation for the Scalar Processing Problem

Program structure is perhaps best characterized by a program graph whose nodes represent basic blocks and whose arcs represent control flow from block to block. A basic block is a maximal set of instructions such that every instruction in the block is executed exactly once each time the block is entered. Efficient concurrent execution is difficult to achieve since basic blocks typically

(i) have few instructions, e.g. three to six[4, 10],

(ii) have internal data dependencies[11], and

(iii) have a branch instruction at the end[12].

Scalar code optimization is typically performed at the block level, but little optimization can be performed with few instructions. Data dependencies limit concurrency in pipelines. Branch instructions have embedded dependencies in the tests for conditional branches and create delay or uncertainty in selecting the next block for execution.

Thus effective optimization techniques must consider a complex of multiple blocks. A convenient representation of such a complex is a decision tree. Without loss of generality, we consider binary decision trees of basic blocks, where each interior block terminates in a two-way conditional branch and each exterior block terminates in an unconditional branch to the root of another decision tree.

Consider the binary decision tree shown in figure 2.1, written in the language C. This simple example is representative of many nonnumeric programs in that conditional

```
struct {
        int A, B, C;
} *x, *y;

if( x →A  <= 1 ) {
        if( x →B  <= 8 ) {
                y →C  = 10;
                goto Z;
        } else {
                x →C  = 9;
                goto Y;
        }
} else {
        if( y →A  <= 2 ) {
                x →C  = 7;
                goto X;
        } else {
                x →B  = 3;
                if( y →B  <= 4 ) {
                        y →C  = 6;
                        goto W;
                } else {
                        x →C  = 5;
                        goto V;
                }
        }
}
```

Figure 2.1. Example of decision tree in source language.

statements are frequently nested and assignment statements are extremely simple. Although in this example the leaves of the decision tree terminate in *goto* statements, they could also represent procedure calls.

In order to discuss performance issues, this source level program fragment must be compiled into machine language. We chose to use a load/store architecture[1, 5, 6, 7, 8, 9] because, by explicitly separating memory references from computations, the compiler has greater flexibility in rearranging instructions to improve pipeline utilization. We also chose to avoid conditional codes by using comparison instructions that produce a boolean value in a register[7], thereby eliminating the constraint that the comparison instruction must immediately precede the conditional branch instruction that uses its result.

An assembly language representation of this program fragment is shown in figure 2.2. Loads are shown as "$a \leftarrow$ A(x)" where the address is specified by base

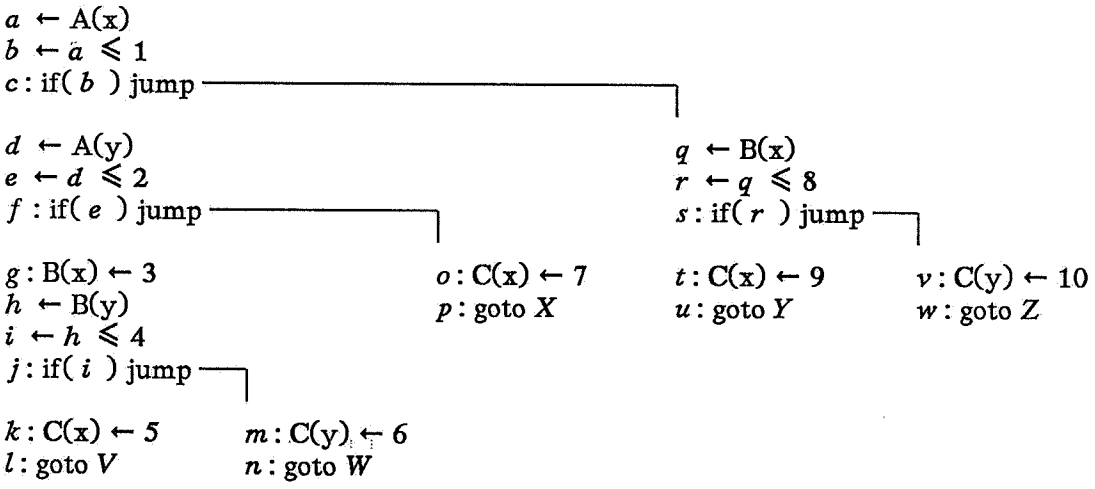

Figure 2.2. Assembly language level representation of decision tree.

register x with displacement A and the content is placed in temporary register a. Similarly, stores are shown as "C(x) ← 5". Comparisons are shown as "$b \leftarrow a \leqslant 1$" where b receives the boolean result of the $a \leqslant 1$ test. Conditional branches on boolean values are shown as "if(b) jump" where the destination of a taken branch is shown by a line. Labels, such as "c:" in the first branch instruction, have been given so that every instruction can be identified either by the result register name or by the label. This notation facilitates understanding of code rearrangements in later examples.

Suppose this code was optimized to run on a machine with a very short instruction pipeline such as the RISC-1 microprocessor[6]. This microprocessor performs a load or delayed branch instruction in two cycles and all other instructions in one cycle. Figure 2.3 shows the execution schedule of the example program on this machine. In this figure the time in clock cycles appears at the left. The completion time for each path through the decision tree is given by the issue time of the pseudo-instruction

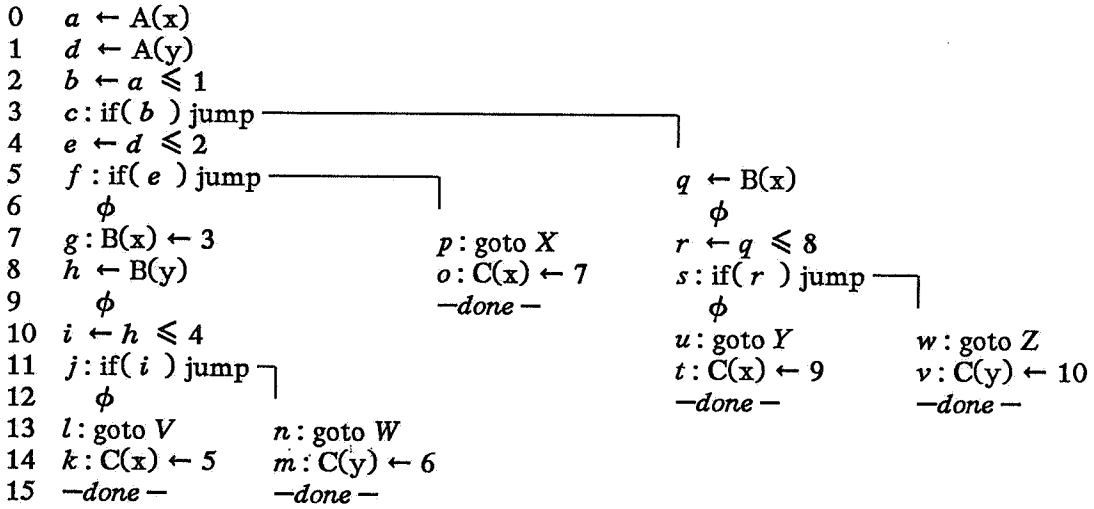

Figure 2.3. Optimum execution schedule for short instruction pipeline.

"*—done—*". Note that a two cycle load instruction, such as d, can be overlapped with another unrelated instruction, b. The load instruction h, however, is followed only by instructions that depend directly or indirectly on the result of the load; thus no instruction can be be overlapped with h and a no-operation instruction, ϕ, must be inserted.

The RISC-1 microprocessor uses delayed branches with length one. In general, a delayed branch with length n means that the n instructions following a branch are always executed regardless of whether the branch is taken[13]. We call the n instructions following a branch the *delayed part* of that branch. Thus instruction e is in the delayed part of branch c. Similarly, instruction k following the unconditional branch l is executed prior to the actual transfer of control.

The code sequence shown in figure 2.3 has been optimized to take maximum advantage of concurrency in the pipeline. The load instruction d has been moved up to fill the delay between the load instruction a and the dependent comparison instruction b. Similarly, the comparison instruction e has been moved up into the delayed part of branch c, and the store instructions k, m, o, t, and v have been moved down into the delayed part of the logically succeeding unconditional branches.

In general, the execution time through a decision tree is dependent on which path is taken through the tree. If the probability of taking path i is p_i and the execution time of path i is t_i, then one measure of performance is the expected execution time

$$\mathrm{E}[T] = \sum_{i=1}^{M} p_i t_i$$

where M is the total number of paths through the decision tree. For figure 2.3, if each path is equally likely, the expected execution time in cycles on a RISC-1 microprocessor is

$$E[\,T_{RISC}\,] = 0.2\,(\,15 + 15 + 9 + 12 + 12\,) = 12.6\ \ (clocks\,)$$

For this example, relatively high utilization (few ϕ cycles) is achieved when the machine has a very short pipeline. On the other hand, short pipelines offer little concurrency, and thus can achieve only relatively low performance. In order to quantify achieved performance, a lower bound on the expected execution time can be derived by assuming an infinite resource dataflow machine. Since data flow uses the assignment of values to trigger dependent instructions, no interior jumps are needed. Thus instructions c, f, j, and s are deleted. However, since the schedule is for one decision tree only, the exterior *goto*'s, l, n, p, u, and w, are retained. With infinite resources, performance is limited only by data dependencies. Using RISC-1 timing, the execution schedule for this example on an infinite resource dataflow machine is as follows.

0	a, d, q
1	
2	b, e, r
3	g, o, p, t, u, v, w
4	h
5	end of paths 3, 4, and 5
6	i
7	k, l, m, n
8	
9	end of paths 1 and 2

The lower bound on the expected execution time is therefore

$$E[\,LB_{RISC}\,] = 0.2\,(\,9+9+5+5+5\,) = 6.6\ \ (clocks\,)$$

The performance ratio is defined as

$$\frac{P_{E[T_{RISC}]}}{P_{LB_{RISC}}} = \frac{\dfrac{1}{E[\,T_{RISC}\,]}}{\dfrac{1}{E[\,LB_{RISC}\,]}} = \frac{E[\,LB_{RISC}\,]}{E[\,T_{RISC}\,]} = \frac{6.6}{12.6} = 52\%$$

Using this metric, the RISC-1 microprocessor achieves a performance level that is only about half of the performance level theoretically possible in an infinite resource

dataflow machine.

Suppose that in an attempt to increase the performance of the machine, the clock speed is increased by a factor of four, causing the instruction pipeline to become four times as long. There is now a possibility of four times the concurrency, but the load and branch instructions take eight cycles to complete while all other instructions take four cycles. Figure 2.4 shows the execution schedule for the same example decision tree. The expected execution time, assuming again that all paths are equally likely, is

$$\mathrm{E}[\,T_{PIPE}\,] = 0.2\,(\,57 + 57 + 36 + 36 + 36\,) = 44.4\ \ (4\times \mathit{clocks}\,)$$

Therefore the speedup is

$$SP_{PIPE} = \frac{4\ \mathrm{E}[\,T_{RISC}\,]}{\mathrm{E}[\,T_{PIPE}\,]} = \frac{50.4}{44.4} = 1.135$$

and the performance ratio is

$$\frac{P_{\mathrm{E}[T_{PIPE}]}}{P_{LB_{RISC}}} = \frac{4\ \mathrm{E}[\,LB_{RISC}\,]}{\mathrm{E}[\,T_{PIPE}\,]} = \frac{26.4}{44.4} = 59\%$$

This speedup is very small in spite of the fact that figure 2.4 has been hand coded for optimal instruction overlap. Note that the speedup calculation for lengthening the pipeline by a factor of four does not take into account the overhead represented by a clock speedup of less than a factor of four due to the additional latching necessary to implement a pipeline with finer granularity[14]. For this example even a small amount of overhead, i.e. a clock speedup of less than 3.524, will cause the speedup to become less than one.

For scalar code, simply increasing the pipeline length is rarely a viable approach to achieving higher performance. The main problem is that, because basic blocks in scalar code tend to be short and contain dependencies, there simply are not enough independent instructions to make use of a high degree of concurrency.

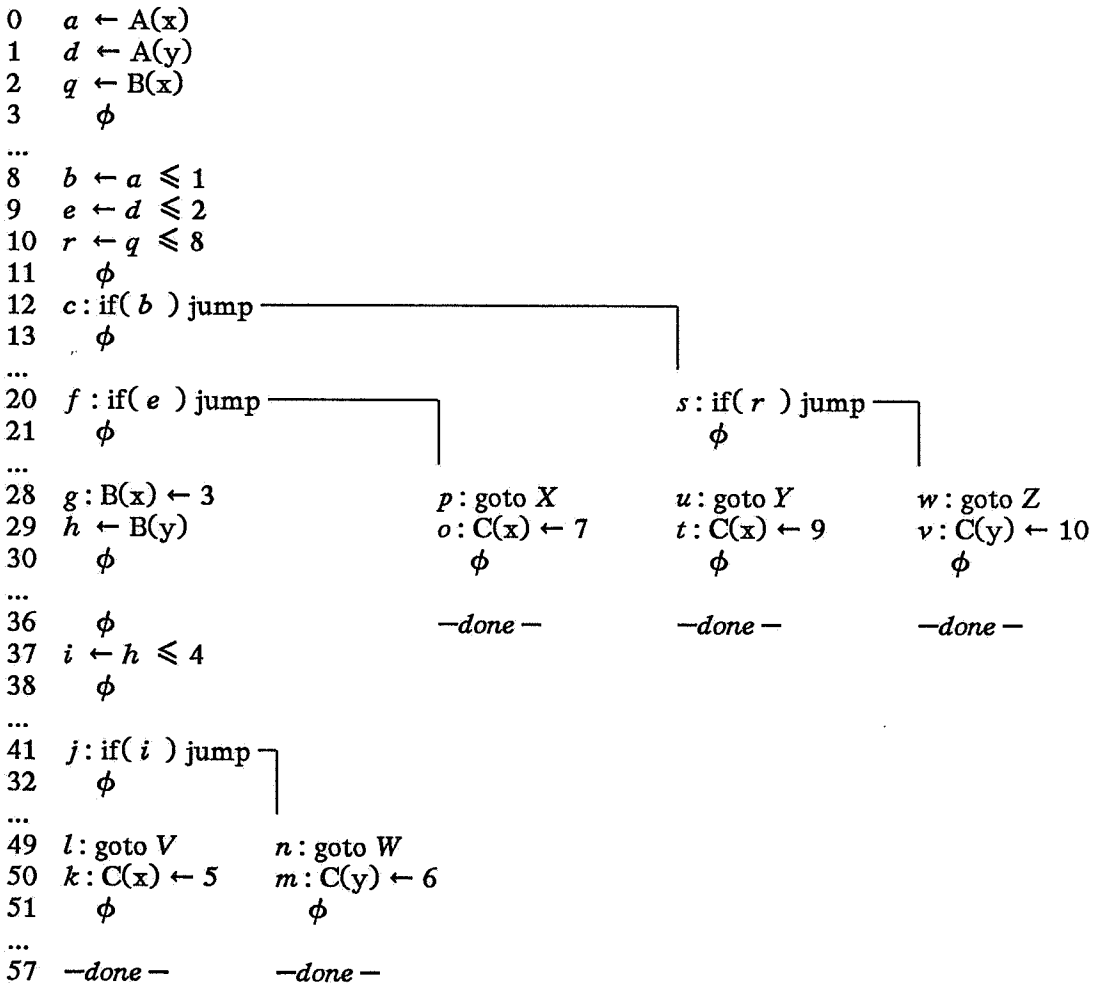

Figure 2.4. Optimum execution schedule for long instruction pipeline.

2.3. Architectural Support for Scalar Processing

In an instruction pipeline, no-operation cycles (ϕ) are inserted either by hardware[1] or by software[8] whenever necessary to insure that data and/or control flow dependencies are met. Some of these dependencies are "real" in the sense that they are inherent in the program as expressed by the programmer. For example, in figure 2.4

the five ϕ's at times 3-7 are necessary because of the dependency through the register a between the instructions at times 0 and 8 and the availability of only two instructions to move up into this gap. On the other hand, some dependencies are an artifact of the architecture. For example, the store instruction g at time 28 will be executed whenever neither of the branches c and f are taken, i.e. whenever b and e are both false. The values of b and e are actually known at time 13, but the store cannot be issued at that time because it must wait for the branches to be completed. Early execution of store instruction g is critical since the relatively slow load instruction h is dependent on g. This dependency arises because the values of x and y are defined external to this decision tree and hence in general the compiler cannot determine whether $x = y$. Therefore to guarantee correctness the compiler cannot allow h to be executed before g. Note that instructions g and h can be issued in consecutive clock cycles if and only if g and h are not dependent and if they reference different memory banks. Otherwise a memory bank conflict occurs at time 29 and the issue of instruction h is delayed by hardware until the conflict is resolved.

One way to avoid delaying a store instruction while waiting for branches to be resolved is to provide a *guard expression*[15] on the store instruction. A guard expression is a boolean valued expression. Whenever a guard expression evaluates to false, it inhibits writing of the final result, thereby converting the instruction being guarded into a no-operation. We represent a guarded store instruction by

<store instruction> ? <guard expression>

where *<guard expression>* is a function of boolean results generated by previously executed comparison instructions. For example, the store instruction g could be changed to the guarded store

g : B(x) ← 3 ? $\bar{e}\&\bar{b}$

The guarded store, g, can now be moved into the delayed part of branch c at time 13, followed by instruction h at time 14. Similarly, jump and *goto* instructions can be guarded and moved up. Conditional branches can be converted to guarded jumps, and other variables can be added to the guard expression. Note that loads, comparisons, and computation instructions need not be guarded, provided that a sufficient number of registers exists.

By making use of guarded stores and guarded jumps, the decision tree shown in figure 2.4 can be further improved to yield the schedule shown in figure 2.5. Note that once the delayed part of a branch has been completed, subsequent guard expressions need not test the value that determined the outcome of that branch. For example, instruction o is executed if $e\&\bar{b}$ is true. However, since the delayed part of branch c is completed at time 19, the compiler uses the fact that if control flows to instruction o, then $b=0$ so there is no need to include the $\bar{b}$ term in the guard expression for instruction o. Similarly, $b=1$ is known if control flows to instructions t and v. Since the delayed part of branch f is completed at time 22, no later instructions need use e in their guard expressions.

The operation of the pipeline for the time interval 19 to 30 is shown in figure 2.6. In this figure, instructions flow through the pipeline from top to bottom. Lower case letters represent instructions from figure 2.5. Upper case letters with subscripts, such as X_k, represent the k^{th} instruction after the label X of a *goto* instruction. A blank entry is used to indicate a ϕ instruction.

Figure 2.6(a) shows the pipeline operation when path 3 of the decision tree is taken. Referring to pipeline segment 8, primed instructions, such as c', indicate a guard expression that evaluates to false. Each of these instructions is converted into no-operation by the hardware. Instructions in parenthesis, such as (h), are fully executed,

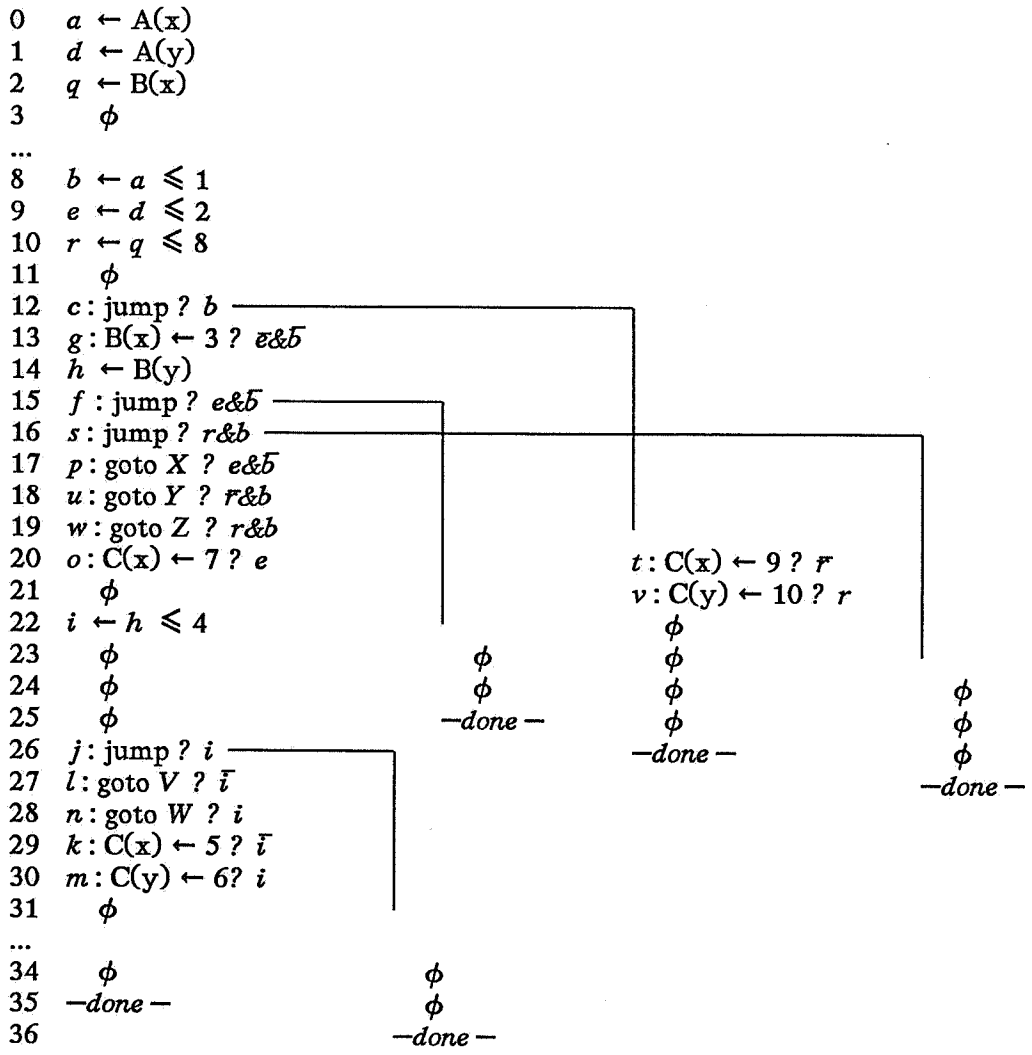

Figure 2.5. Improved schedule using guarded store and jump instructions.

but are of no use to this path. Note that instructions g', (i), and o on path 3 complete early because they are four-cycle instructions.

Figure 2.6(b) shows the pipeline operation when path 5 is taken. Note that although the same set of instructions as for path 3 are completed at segment 8 between times 19 and 26, their guard expression values are different and hence a different set of

PIPELINE SEGMENT	TIME 19	20	21	22	23	24	25	26	27	28	29	30
1	w	o		i			X_0	X_1	X_2	X_3	X_4	X_5
2	u	w	o		i			X_0	X_1	X_2	X_3	X_4
3	p	u	w	o		i			X_0	X_1	X_2	X_3
4	s	p	u	w	o		(i)			X_0	X_1	X_2
5	f	s	p	u	w						X_0	X_1
6	h	f	s	p	u	w						X_0
7		h	f	s	p	u	w					
8	c'		(h)	f	s'	p	u'	w'				

(a) path 3 (b =0, e =1)

PIPELINE SEGMENT	TIME 19	20	21	22	23	24	25	26	27	28	29	30
1	w	t	v						Z_0	Z_1	Z_2	Z_3
2	u	w	t	v						Z_0	Z_1	Z_2
3	p	u	w	t	v						Z_0	Z_1
4	s	p	u	w	t'	v						Z_0
5	f	s	p	u	w							
6	h	f	s	p	u	w						
7		h	f	s	p	u	w					
8	c		(h)	f'	s	p'	u'	w				

(b) path 5 (b =1, r =1)

Figure 2.6. Pipeline operation showing overlapped guarded jumps.

these instructions is converted to no-operations. Once time 23 is reached instructions exiting the pipeline on the two paths are distinct, since the outcome of branch c differs at time 19, resulting in different instructions being initiated at time 20 and these instructions appear at segment 4 at time 23.

From figure 2.5 it is clear that the delayed parts of branches are utilized much more efficiently. The expected execution time and speedups using guarded instructions, assuming equal probability paths, are

$$E[\,T_{GUARD}\,] = 0.2\,(\,35 + 36 + 25 + 26 + 27\,) = 29.8\ \ (4\times \mathit{clock}\,)$$

$$SP_{GUARD:\,RISC} = \frac{4\,E[\,T_{RISC}\,]}{E[\,T_{GUARD}\,]} = \frac{50.4}{29.8} = 1.691$$

$$SP_{GUARD:\,PIPE} = \frac{E[\,T_{PIPE}\,]}{E[\,T_{GUARD}\,]} = \frac{44.4}{29.8} = 1.490$$

Although the achieved speedup of 1.691 relative to a 1× pipeline is still much less than the theoretical 4× speedup, it is relatively near the data flow limit, as shown by the performance ratio

$$\frac{P_{E[\,T_{GUARD}\,]}}{P_{LB_{RISC}}} = \frac{4\,E[\,LB_{RISC}\,]}{E[\,T_{GUARD}\,]} = \frac{26.4}{29.8} = 89\%$$

Recall that this performance ratio for a 4× pipeline without guarded instructions is only 59%.

The amount of hardware required to evaluate guard expressions depends on the complexity of the expressions. We have found that expressions consisting of the *AND* of true and complemented values of a few register bits is adequate for supporting fast scalar processing. Thus very fast guard expression evaluation can be implemented inexpensively.

To exploit the guarded store and guarded jump instructions discussed above, it is necessary to perform extensive code rearrangement. Constraints on code rearrangement arise from data dependencies between instructions, hence it is critical that artificial dependencies are eliminated whenever possible. An important class of artificial dependencies arise due to register reuse. In the following example, no parallelism can be exploited in the code sequence on the left because the instructions forms a dependency chain.

$a \leftarrow x + y$
$b \leftarrow a + z$
$a \leftarrow u + v$
$c \leftarrow a + w$

$a \leftarrow x + y$
$b \leftarrow a + z$
$a' \leftarrow u + v$
$c \leftarrow a' + w$

The code sequence on the right, however, forms two independent dependency chains and thus execution of one chain can be overlapped with the other. The improvement in parallelism was achieved by renaming the second assignment of register a so as to avoid reuse. The technique of register renaming to eliminate unnecessary dependencies is well known[16]. When applied to a tree of basic blocks, every temporary register can be renamed so as to be assigned exactly once, the so called *single assignment* property.

The use of single assignment temporaries gives the compiler maximum flexibility in reordering code within a decision tree so as to utilize concurrent resources efficiently. All code shown in the examples has this single assignment property. Note that in actuality some register reuse can be accommodated without performance degradation. Once a code sequence has been generated, a second pass can be made over the generated code to locate disjoint uses of registers and map them into the same physical register. No performance is lost by this mapping and register requirements are reduced.

Architectural support for register renaming to increase parallelism simply involves providing a sufficiently large number of registers. Extensive code rearrangement, however, poses a more serious problem. When code is moved from after a branch to before the branch, some instructions from the conditionally taken and fall-through paths of the branch become unconditionally executed. In this case a spurious exception condition can occur in the rearranged code due to the execution of an instruction that would not have been executed in a serial machine.

One possible solution is to encode the exception condition within the result register[17], possibly by extending the length of the register. Exception conditions are propagated through subsequent computations, but the actual signaling of exception

conditions is deferred until an attempt is made to store the content of a register containing an exception code. With rearrangement, the set of store instructions not inhibited by guard expressions are exactly the same as the set of stores produced by a serial execution of the program without rearrangement. Thus the signalled exceptions are the same. The same encoding technique as for arithmetic exceptions such as divide by zero can be used for illegal memory references so as to allow actions like prefetching past the end of an array. However this technique cannot easily be extended to page fault exceptions, since they may eventually have to be serviced.

2.4. The Decision Tree Scheduling Technique

In the previous section we proposed the guarded store and guarded jump architectural features. These architecture features can significantly improve scalar code performance. Efficient use of these features, however, requires that the compiler perform extensive code rearrangement. This section describes a heuristic code generation technique that performs the necessary code rearrangement.

The main objective of the code generator is to rearrange, i.e. *schedule*, the instructions in a decision tree so as to minimize the expected execution time through the decision tree. The technique presented here was used to produce the schedule shown previously in figure 2.5. A convenient representation of the program for the purpose of instruction scheduling is the dependency graph[16]. Figure 2.7 shows the dependency graph for the ongoing decision tree example. Instructions are shown as nodes in the graph, labeled with either the result register name or the explicit instruction label. Arcs in the graph represent data or control dependencies between instructions. Within each node, the number on the left hand side of the center row gives the earliest time that the node can be issued. The number on the right hand side gives the execution delay of the node. Note that the delay of jumps in the interior of the tree (c, f, j, s) have

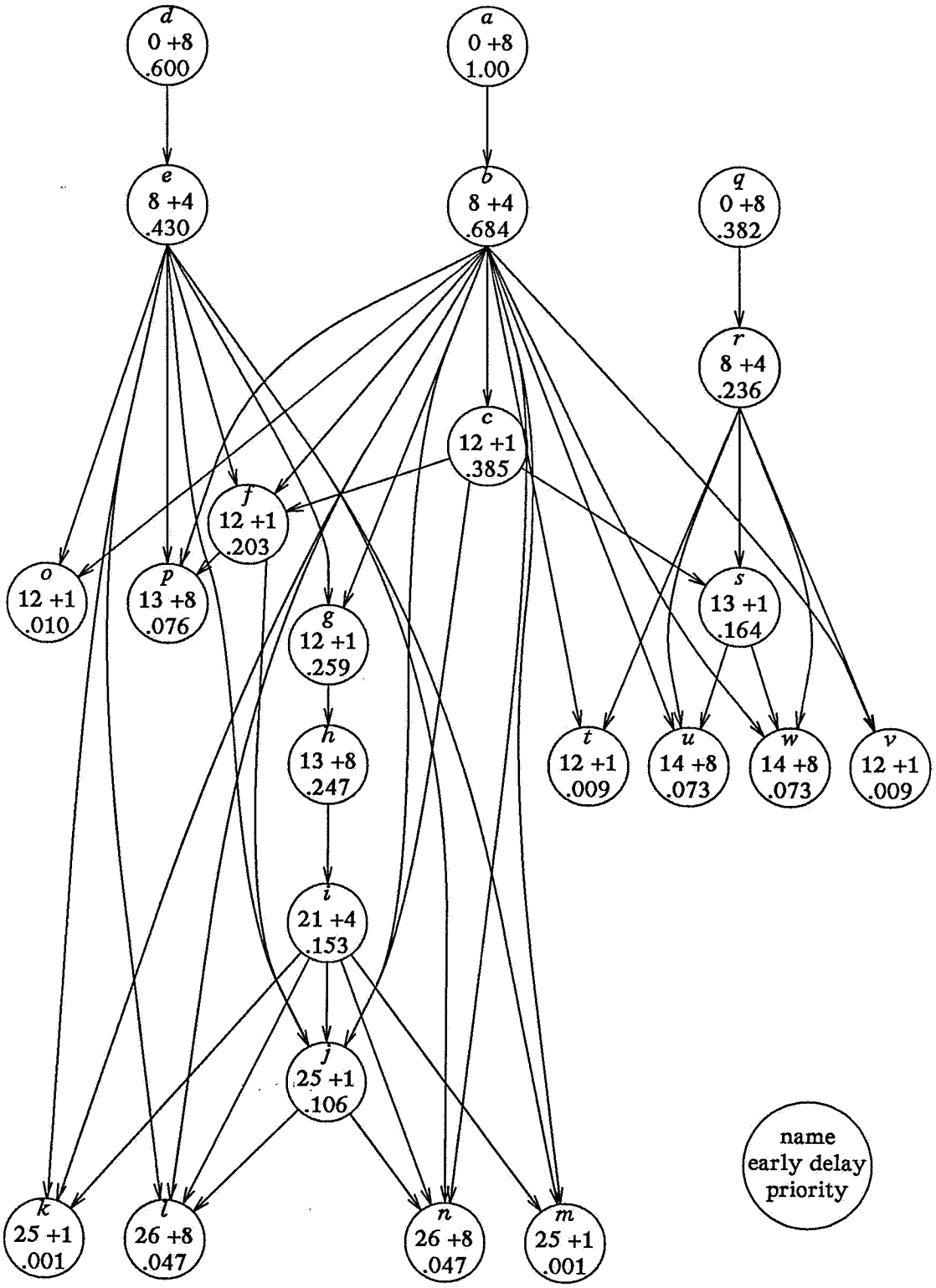

Figure 2.7. Dependency graph representation of decision tree.

execution delay +1 to allow overlapped jumps while leaf jumps (l, n, p, u, w) have delay +8 in our model since this tree must be completed before the next tree begins. The real number at the bottom of the node is the scheduling priority, to be discussed later.

The problem of finding a minimum delay schedule for a set of dependent tasks is known to be NP-hard[18]. However, it is well known that list scheduling techniques[19] produce good schedules in practice. We have developed a decision tree scheduling (DTS) technique based on an extension of list scheduling. A procedure implementing the DTS technique is shown in figure 2.8. This procedure is initially invoked with G equal to the entire dependency graph and P equal to the set of all paths through the decision tree from the root to a leaf node. On the initial call, nothing is deleted from the graph in step 1 since every node in the graph lies on at least one path

---

procedure *schedule*(G, P)

G: Dependency graph representing subprogram to be scheduled

P: Set of paths through subtree to be scheduled

begin

1. Delete from G those nodes and arcs not on paths in P
2. Return if G is empty
3. Schedule nodes until potential transfer of control flow
4. *schedule*(G, {jump taken paths})
5. *schedule*(G, {jump not taken paths})

end.

Figure 2.8. Decision tree scheduling procedure.

---

through the tree. Step 3 schedules code in priority order subject to dependency constraints. Code scheduling continues until n cycles after the first interior branch has been scheduled, where n is the delay of the branch. At this time two logically independent subtasks are created to handle the two possible branch outcomes and *"schedule"* is called recursively.

The first subtask is initiated with a copy of the dependency graph along with the subset of paths through the decision tree that pertains if the jump is taken (step 4). Similarly, the second subtask is initiated with a copy of the dependency graph and the subset of paths that pertain if the jump is not taken (step 5).

Each subtask schedules code until n cycles after the next interior jump that belongs to its own subset of paths. Note that this jump may have been scheduled by the parent task in the delayed part of some earlier jump. Referring to figure 2.5, the subtask handling the code sequence for paths $\{4, 5\}$ beginning with instruction t finds previously scheduled jump s to belong to paths $\{4, 5\}$ but not jump f, since f belongs to paths $\{1, 2, 3\}$. Therefore the subtask for $\{4, 5\}$ would stop code scheduling at time $x_s + n = 16 + 8 = 24$, where x_s is the issue time of instruction s, and recursively divide into two subtasks to complete paths $\{4\}$ and $\{5\}$ independently. Note that exterior jumps (*goto*'s) do not terminate code scheduling in a task. The recursive division continues to the leaves of the decision tree. Application of this code scheduling procedure to the dependency graph shown in figure 2.7 was used to produce the code shown in figure 2.5.

The quality of code generated by the DTS technique is dependent on the heuristic used to compute the node priorities. Intuitively, the node priorities should satisfy the following properties:

(1) A node on a high probability path through the decision tree should be given higher priority than a node on a low probability path. Since a node can be on multiple paths, the priority of a node should depend on the sum of the path probabilities.

(2) On a given path, a node near the top of the critical path through the dependency graph should be given higher priority than a node near the bottom of the critical path. Also, a node not on the critical path should be given lower priority than a node on the critical path[20].

Property two can be quantified as follows: Focus on a single path i through the decision tree. Calculate, for each node on that path, the earliest execution time based on data dependencies. Define the path length l_i to be the earliest completion time of the terminal *goto* instruction of path i. Sweep backward through the graph and calculate, for each node j, the latest time ($latest_j$) that each node must be issued in order for the path to be completed within the minimum time l_i. Nodes on which the terminal *goto* instruction does not depend have latest issue time of l_i minus the execution time of that node. Define the *urgency* of a node j on path i through the decision tree to be

$$u_{i,j} = 1 - \frac{latest_j}{l_i}$$

Figures 2.9, and 2.10 show the earliest issue time, the latest issue time, and the urgency for each node on each path through the decision tree.

Combining the urgency metric with property one gives the heuristic priority function. For each node j, the list scheduling priority w_j is given by

$$w_j = \sum_{i=1}^{M} p_i u_{i,j}$$

where M is the number of paths through the decision tree and p_i is the probability of taking path i. Application of this heuristic function to the urgency values shown in

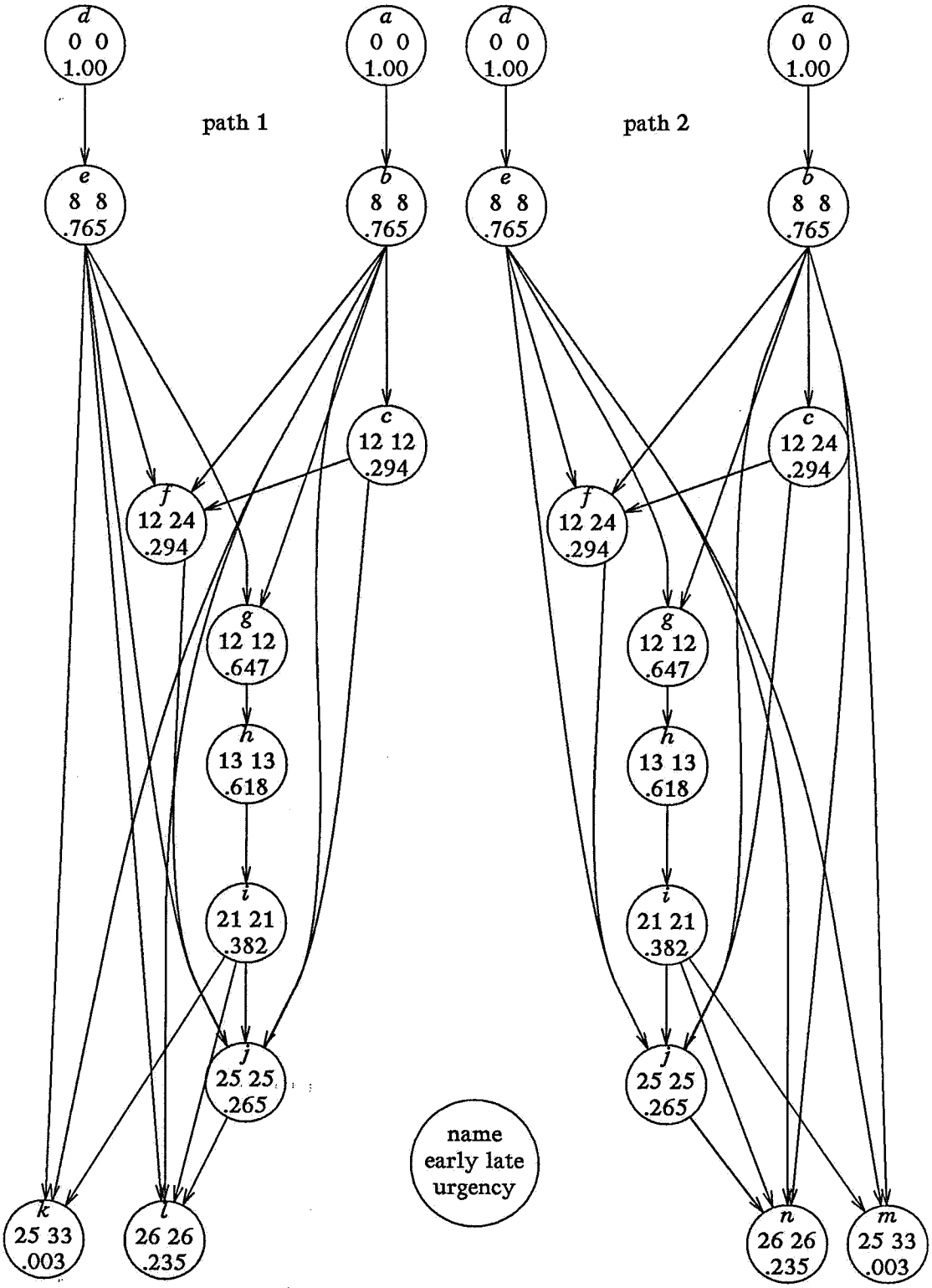

Figure 2.9. Heuristic computation for paths 1 and 2.

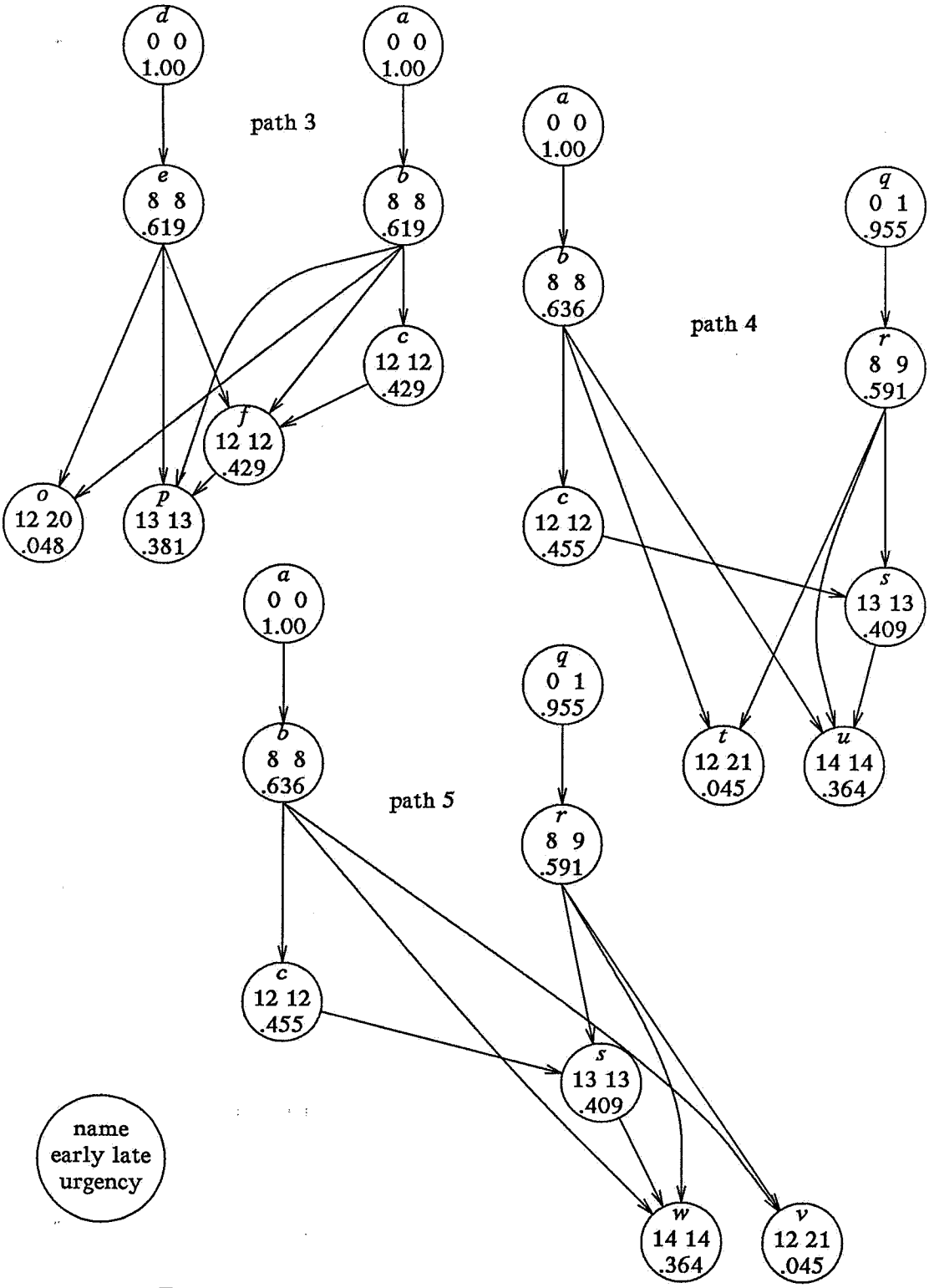

Figure 2.10. Heuristic computation for paths 3, 4, and 5.

figures 2.9 and 2.10, where each p_i is assumed to be 0.2 in this example, was used to provide the priority values shown in figure 2.7.

A significant advantage of the DTS technique is the sensitivity of the heuristic to the values of the path probabilities. For example, if instead of being equal, the path probabilities were (1/8, 1/8, 1/8, 1/2, 1/8), then the new node priorities would be as shown in table 2.1. Based on these new priority values, the DTS technique would produce the new schedule shown in figure 2.11. Note that path 4 (the high probability path) has been shortened from 26 to 22 cycles at the expense of increasing the length of paths 1, 2, and 3. Table 2.2 shows the expected execution time and speedup relative to the RISC-1 processor for each of the schedules under the asymmetric path probability assumption. The RESCHED column uses the schedule in figure 2.11 and reflects the advantage of rescheduling when path probabilities change.

Table 2.1. Node priorities for path probabilities (1/8, 1/8, 1/8, 1/2, 1/8).

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*
1.000	0.666	0.412	0.375	0.269	0.127	0.162	0.155
i	*j*	*k*	*l*	*m*	*n*	*o*	*p*
0.082	0.066	0.000	0.029	0.000	0.029	0.006	0.048
q	*r*	*s*	*t*	*u*	*v*	*w*	
0.597	0.369	0.256	0.023	0.182	0.006	0.046	

Table 2.2. Expected execution time and speedup for asymmetric probabilities.

	RISC	PIPE	GUARD	RESCHED
E[*T*]	49.500	41.250	28.375	27.000
$SP_{:RISC}$		1.200	1.744	1.833

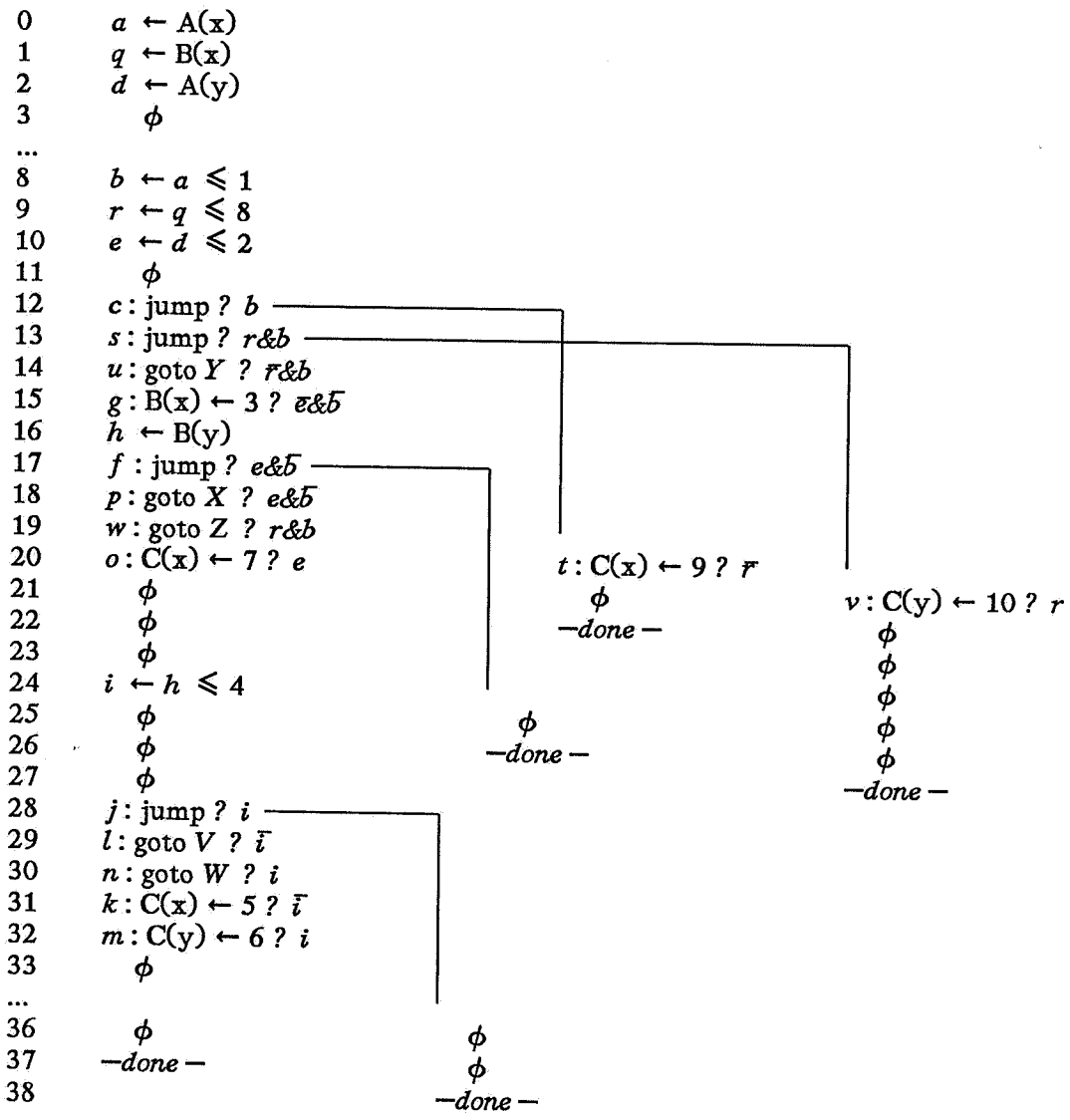

Figure 2.11. Execution schedule for path probabilities (1/8, 1/8, 1/8, 1/2, 1/8).

The sensitivity of the proposed heuristic to the actual path probability can be viewed as a refinement of the trace scheduling technique[21], which generally does not distinguish between relatively equal path probabilities verses highly biased probabili-

ties. Alternatively, the DTS technique can be viewed as an extension of the IF-tree technique proposed by Davis[22]. An IF-tree is a binary decision tree transformed to use a large multiway branch. Since a multiway branch cannot be executed until all dependent conditions have been evaluated, exits from the decision tree cannot be made until the last condition has been evaluated. In contrast the DTS technique takes advantage of early exits out of the decision tree whenever possible.

The DTS technique can be extended to architectures that employ parallel instruction pipelines and a horizontal microcode-like instruction format so as to permit issuing multiple instructions per clock cycle. For example, instead of increasing the RISC-1 clock speed by a factor of four and hence multiplying the pipeline length by four as shown in figure 2.5, the same level of concurrency can be achieved by multiplying the pipeline length by two and then duplicating the pipeline. Using the same node priorities as shown in figure 2.7, the new schedule with up to two instructions issued per cycle is shown figure 2.12. In this figure the instructions are shown by their label and guard expression. The top of each box represents the branch target line used in previous figures. When the compiler generates two guarded jumps in the same clock cycle, they must be mutually exclusive so the hardware implementation remains simple. Assuming equal path probabilities, the expected execution time, speedup, and performance ratio are

$$E[\,T_{PAR-2}] = 0.2\,(\,17 + 18 + 12 + 13 + 13\,) = 14.6 \;\; (2\times \mathit{clock}\,)$$

$$SP_{PAR-2:\,RISC} = \frac{2\,E[\,T_{RISC}\,]}{E[\,T_{PAR-2}]} = \frac{25.2}{14.6} = 1.726$$

$$\frac{P_{E[\,T_{PAR-2}]}}{P_{LB_{RISC}}} = \frac{2\,E[\,LB_{RISC}\,]}{E[\,T_{PAR-2}]} = \frac{13.2}{14.6} = 90\%$$

This schedule with a 2× clock achieves a speedup slightly better than the 1.691 speedup obtained by the single pipeline with a 4× clock.

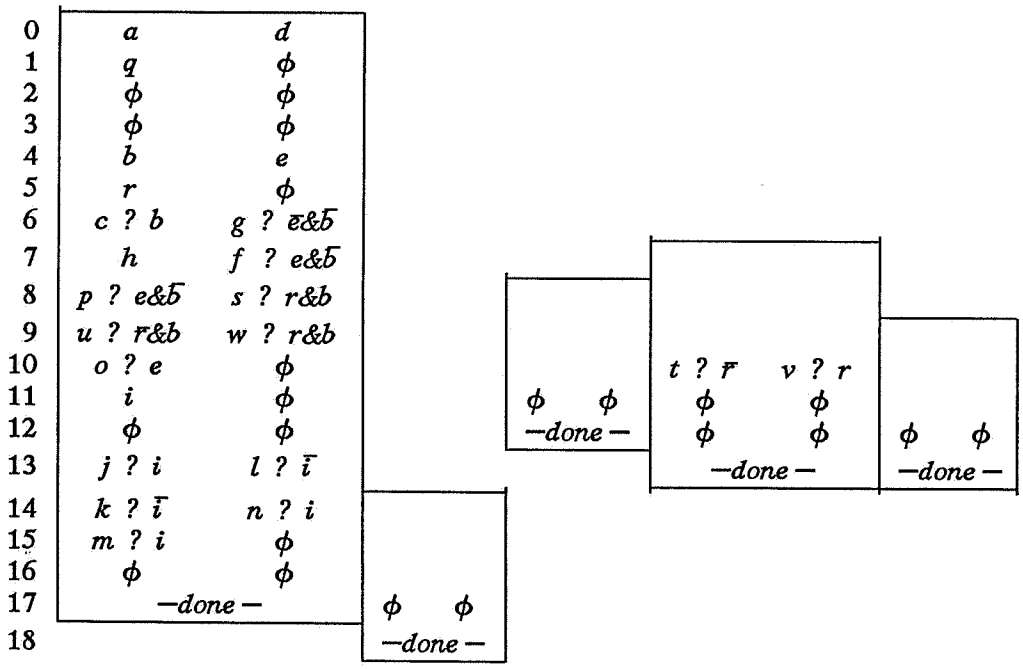

Figure 2.12. Execution schedule for two instructions per cycle.

Another configuration with the same concurrency is 4 RISC-1 pipelines using a 1× clock. The schedule for this configuration is shown in figure 2.13. In this figure the guard expressions have been omitted to save space. Again assuming equal path probabilities, the expected execution time, speedup, and performance ratio are

$$\mathrm{E}[\,T_{PAR-4}] = 0.2\,(\,9+9+6+6+6\,) = 7.2\ \ (1\times \mathit{clock}\,)$$

$$SP_{PAR-4:\,RISC} = \frac{\mathrm{E}[\,T_{RISC}\,]}{\mathrm{E}[\,T_{PAR-4}]} = \frac{12.6}{7.2} = 1.750$$

$$\frac{P_{\mathrm{E}[\,T_{PAR-4}]}}{P_{LB_{RISC}}} = \frac{\mathrm{E}[\,LB_{RISC}\,]}{\mathrm{E}[\,T_{PAR-4}]} = \frac{6.6}{7.2} = 92\%$$

The ability of the DTS heuristic to utilize both pipelining and parallelism efficiently significantly increases the flexibility of the machine organization and allows the machine

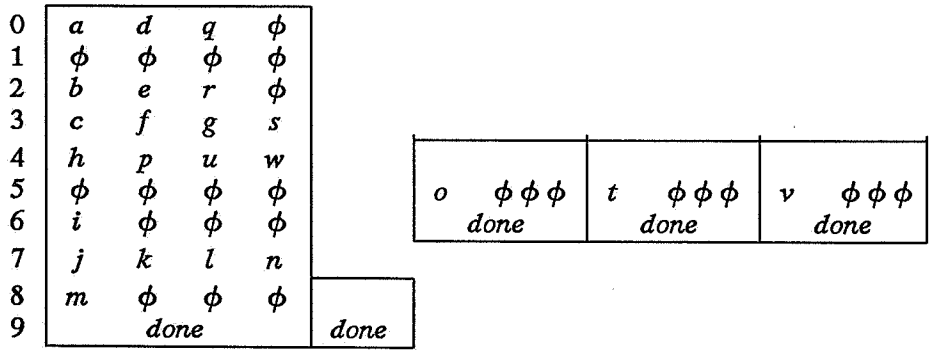

Figure 2.13. Execution schedule for four instructions per cycle.

designer to trade off pipelining with parallelism for greater cost-effectiveness.

2.5. Performance Evaluation

We have constructed a compiler to evaluate the performance of the DTS technique and the guarded store and jump architectural features. This compiler accepts a subset of the language C.

The performance evaluation is based on a pipelined uniprocessor model derived from the scalar portion of the Cray-1. This baseline uniprocessor has instruction execution timing characteristics of the Cray-1 computer[23, 24] with branches taking a constant 14 cycles. The Cray-1 branch time is actually 2, 5, or 14 clocks for the cases that the branch is not taken, taken with branch target in instruction buffer, or taken with branch target in main memory, respectively. The constant 14 clock assumption simplifies the baseline uniprocessor, but makes its performance somewhat lower than the performance of the actual Cray-1 on scalar code.

In this section *performance* is speedup relative to this baseline uniprocessor. The target machine model consists of one or more pipelined processing elements. The

processing elements are controlled by a horizontal instruction word with an instruction field for each processing element. The timing characteristics of the processing element are the same as that of the baseline uniprocessor.

In addition to the normal operation of code generation, our compiler also performs selective code replication. Use of replication was motivated by the observation that pipeline utilization improves with larger decision trees. Natural decision trees in most programs tend to be very small because every basic block that has more than one predecessor blocks becomes the root of a distinct decision tree. For example, the program fragment shown in figure 2.14(a) produces three decision trees. The first tree contains blocks c_1, S_1, and S_2, the second tree contains blocks c_2, S_3, and S_4, and the third tree contains the block S_6.

By replicating the second if-statement and statement S_6, a single much larger decision tree can be produced as shown in figure 2.14(b). This tree consists of the 7 blocks

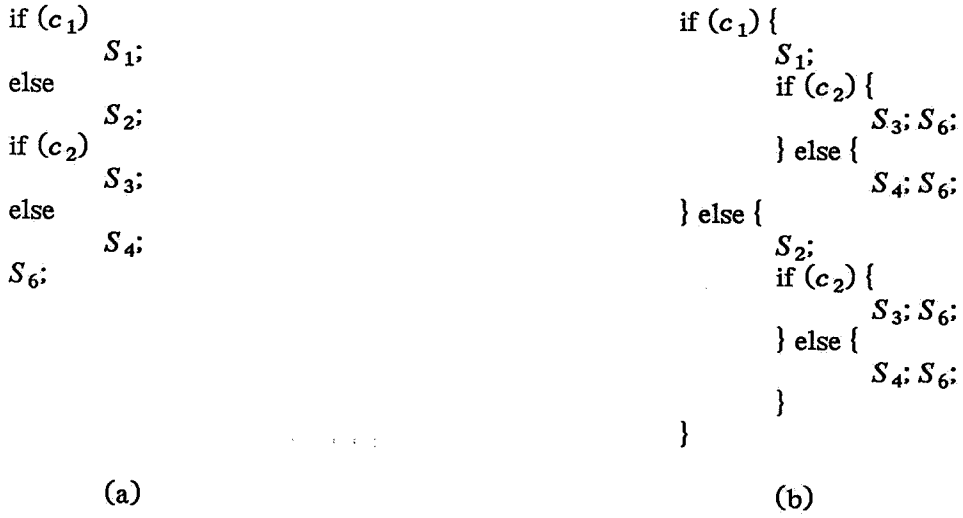

Figure 2.14. Use of code replication to produce larger decision trees.

c_1, (S_1, c_2), (S_3, S_6), (S_4, S_6), (S_2, c_2), (S_3, S_6), and (S_4, S_6).

In our compiler, code replication is controlled by a parameter ϵ. As long as a path of a decision tree has probability greater than ϵ, an attempt is made to replicate code further along that path until conditional branches cause the path probability to fall below ϵ. This technique of weighted code replication is advantageous in that high probability subtrees of a decision tree are made deeper and larger by code replication while low probability subtrees are kept small.

Table 2.3 shows the achieved performance for a binary search program with the list to be searched represented as a balanced linked binary tree with integer keys. In this table, p is the number of processing elements, i.e. the number of instructions that can be issued in parallel each clock cycle, and ϵ is the level of code replication. Each successive column represents code replication past one more conditional branch.

Table 2.3 shows that relatively good speedup of 2.392 can be achieved by the DTS technique with guarded operations on a uniprocessor (p =1) and no code replication (ϵ=1.0). Additional speedup can be achieved either by code replication (going right) or by increasing the level of parallelism in the machine (going down). Greater speedup can be achieved by combining both code replication and increased parallelism.

Table 2.3. Performance of linked binary tree search.

	ϵ					
p	1.0	0.5	0.25	0.125	0.062	0.031
1	2.392	3.116	3.477	3.695	3.782	3.895
2	2.739	3.651	4.107	4.409	4.587	4.781
3	2.831	3.881	4.409	4.707	4.953	5.137
4	2.831	4.042	4.567	4.911	5.174	5.366
∞	2.831	4.141	4.896	5.388	5.733	5.989

Table 2.4 shows the achieved performance for a quicksort algorithm. Although this algorithm is much more complex than the linked binary tree search example, similar speedup trends are observed. The initial (p=1, ϵ=1) speedup is much lower, but

Table 2.4. Performance of quicksort algorithm.

	ϵ					
p	1.0	0.5	0.25	0.125	0.062	0.031
1	1.489	2.396	2.787	2.981	3.021	3.121
2	1.623	2.940	3.391	3.814	3.981	4.120
3	1.644	3.119	3.764	4.193	4.388	4.571
4	1.657	3.176	4.026	4.461	4.683	4.873
∞	1.664	3.311	4.388	5.412	5.904	6.204

Table 2.5. Performance of vmsched.c from the UNIX kernel.

	ϵ				
p	1.0	0.5	0.25	0.125	0.062
1	1.713	1.863	1.953	2.049	2.108
2	2.127	2.456	2.626	2.846	2.992
3	2.307	2.727	2.915	3.292	3.489
4	2.373	2.866	3.089	3.538	3.786
∞	2.431	3.028	3.456	4.177	4.658

Table 2.6. Performance of hydro excerpt from Livermore benchmarks.

	ϵ				
p	1.0	0.997	0.992	0.99	0.98
1	1.313	2.358	3.052	3.223	3.416
2	1.406	2.742	4.895	5.632	6.468
3	1.441	2.742	4.999	5.985	9.021
4	1.441	2.742	4.999	5.985	9.212
∞	1.441	2.742	5.000	5.986	9.217

reducing ϵ has even higher relative payoff. Once ϵ is reasonably low, increasing p has a much greater effect.

Both of the above examples are very small program fragments. To test the viability of our techniques on more difficult examples, we evaluated a number of program modules from the Berkeley UNIX kernel. Table 2.5 shows the achieved performance for the virtual memory scheduler from the UNIX kernel. Less speedup was achieved relative to the previous examples because this program module contains many procedure calls to externally defined procedures. An external procedure call always terminates a path in the decision tree since code replication cannot proceed without knowledge of the called procedure. Hence the full power of code replication could not be applied in this example. Nevertheless, even this example demonstrates that our approach is capable of delivering significant speedup on branch-intensive scalar code.

We have found that our DTS compilation technique and the guarded store and jump architectural features are very effective with parallel-pipeline hardware for vectorizable code as well as scalar code. Table 2.6 shows the achieved performance for the first loop from the Lawrence Livermore Loops benchmarks[25]. On a uniprocessor with the loop unrolled once via code replication (p =1, ϵ=0.997), the result is comparable to hardware speedup techniques for scalar processors[26]. Using three processing elements and unrolling the loop four times (p =3, ϵ=0.98), DTS produces a speedup of 9.021 via a schedule whose average execution time is 2.62 cycles per loop iteration. In contrast, since the Cray-1 has only a single floating-point multiply pipeline and this example uses 3 multiplications per loop iteration, the maximum Cray-1 performance in vector mode is no fewer than 3.00 cycles per loop iteration, i.e. a maximum speedup of 7.883 relative to its scalar mode performance.

CHAPTER 3

SCHEDULING SIMPLE LOOPS FOR OPTIMAL THROUGHPUT

3.1. Introduction

In a conventional programming environment many programs spend a large fraction of their execution time in looping constructs. Therefore the optimization of program loops in order to speed up their execution time is of paramount importance in a high performance computer system.

Although loops can be viewed as scalar code and scheduled to achieve higher performance using the decision tree scheduling (DTS) technique described in chapter 2, in general the DTS technique cannot deliver maximum loop performance since the DTS technique has the restriction that a decision tree complete execution before another decision tree can begin. This strategy allows distinct trees to be scheduled independently and was deemed necessary in order to reduce the complexity of the scheduling problem to a manageable level. A disadvantage of scheduling trees independently is that performance is compromised during the transition from one tree to another. Therefore loop performance is degraded if tree transitions occur while the loop is being executed.

By using code replication to cause loop unrolling, the number of tree transitions can be reduced since multiple iterations of the loop can be executed by a single decision tree. However tree transitions can never be totally eliminated unless the loop is completely unrolled. Complete loop unrolling is rarely feasible since loop iteration limits are frequently data-dependent and/or the number of iterations is so large that complete unrolling is impractical. Therefore in general some performance degradation is inevitable when loops are scheduled using the DTS technique.

The structure of general loops involving nested conditional statements and procedure calls are quite complex, and perhaps in practice it is best to handle them using the DTS technique in spite of the fact that tree transitions lead to suboptimal performance. However, simple loops whose bodies consist solely of assignment statements have a very simple and regular structure. This structural regularity can be exploited to unroll the loop logically and completely without actually doing so. This is the basis for the *simple loop scheduling* (SLS) technique proposed in this chapter.

The DTS technique is a general technique that is applicable to any program construct. The SLS technique, on the other hand, is applicable only to a restricted class of loops. The advantage of the SLS technique is that it produces schedules that are *throughput optimal*, i.e. optimal performance is maintained as long as the loop continues to iterate. Suboptimal performance occurs only when the loop starts and when it terminates.

The importance of high-speed loop execution is well known. Since loop speedup techniques depend on the machine model chosen, we begin this chapter with a brief review of several well-known machine models. Following that, the SLS technique is developed in several stages, beginning with simple cases.

3.2. Architectures and Loop Performance

Consider the simple loop written in the language C shown in figure 3.1. This program fragment is representative of many nonnumeric programs in that simple loops are frequently used to traverse linked data structures. The C notation

while (<*assignment* >) { ... }

means perform the <*assignment* > statement as specified and retain the value that was assigned. Then if the retained value is nonzero (i.e. if the pointer is valid), initiate

```
struct vertex {                    /* Graph vertex descriptor. */
   ...                             /* Other fields in vertex descriptor. */
   struct arc **arc;               /* Pointer to list of outbound arc descriptors. */
   int data;                       /* A data field of interest. */
};

struct arc {                       /* Graph arc descriptor. */
   ...                             /* Other fields in arc descriptor. */
   struct vertex *node;            /* Pointer to destination vertex. */
};

int T[ ];                          /* A list of indices into the following array. */

struct {                           /* List of records in an array. */
   int index ;                     /* A value used to index into the array T. */
   int value ;                     /* An interesting data value. */
} R[ ];

register struct vertex *c ;        /* A cursor moving through the graph. */
register int i = 0;                /* A corresponding index into the array T. */
register int g ;                   /* A temporary used to index into R. */
register int k = 0;                /* An accumulator of interesting values. */

while (c = c →arc[1]→node) {       /* While more vertices do: */
   g = T[c →data + i ];            /* Calculate an index into R. */
   i = R[g ].index ;               /* Acquire the next corresponding index. */
   k += R[g ].value ;              /* Accumulate another interesting value. */
}
```

Figure 3.1. Example of simple loop in source language.

another iteration of the loop body. If the retained value is zero, the loop terminates. Note that the <*assignment* > statement is reexecuted at the beginning of each iteration of the loop.

The data structures and operational characteristics of this simple loop are shown in figure 3.2. The top part of the figure shows a graph data structure. The graph is traversed by following the second outbound arc from each vertex (array indices begin with 0 in the language C). The bottom part of the figure shows a pair of related tables.

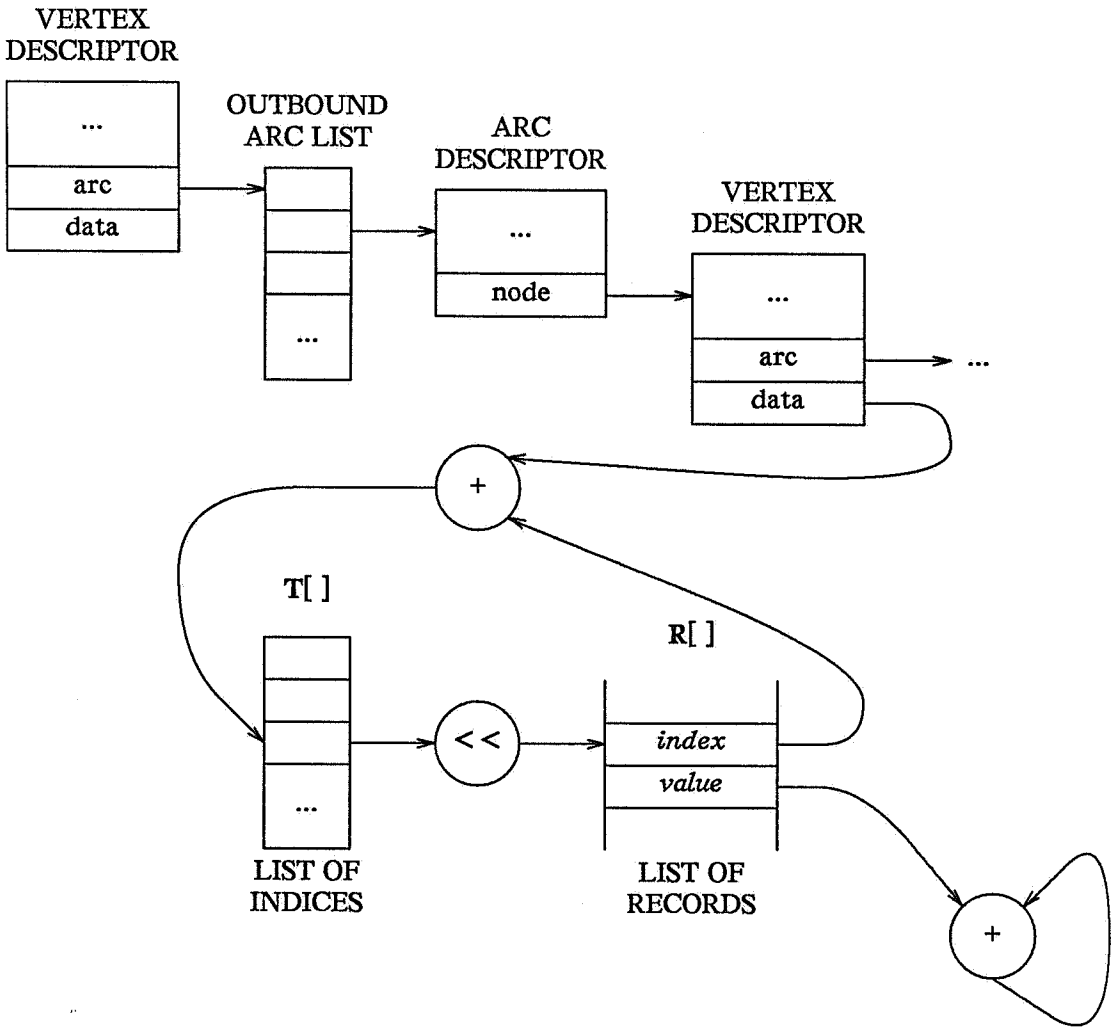

Figure 3.2. Data structures in simple loop.

During each iteration of the loop, an index into the first array, **T**, is generated using an item from the current vertex in the graph. The value retrieved from table **T** is shifted by the appropriate amount so as to form an offset into the second array, **R**. The entry pointed to in **R** (an *index* field) is saved in i to be used during the computation of the

next index into table **T**. The next entry in **R** is the *value* field to be accumulated in the variable k. The final value of k is assumed to be used eventually outside of the loop.

An assembly level representation of this simple loop is shown in figure 3.3. We have chosen to use a load/store architecture[1, 5, 6, 7, 8, 9] for reasons discussed in chapter 2. From this low level representation of the program we can derive the dependency graph shown in figure 3.4. Instructions are shown as nodes in the graph, labeled with either the result register name or the explicit instruction label. Solid arcs represent data dependencies between nodes; downward arcs specify dependencies within a single iteration of the loop while upward arcs specify dependencies from one iteration to the next. Dashed arcs represent control dependencies resulting from conditional branches. The dependency graph is an appropriate representation of a program loop for the purpose of readily viewing the constraints on code scheduling. In future examples we shall omit the lengthly process of specifying program fragments and simply use only dependency graphs.

loop:	$a \leftarrow$ arc(c)	Load pointer to arc pointer list.
	$b \leftarrow 1(a)$	Load pointer to second arc descriptor.
	$c \leftarrow$ node(b)	Load pointer to destination of arc.
	d: if (c=0) *exit*	Terminate loop if no more vertices.
	$e \leftarrow$ data(c)	Load data field from vertex descriptor.
	$f \leftarrow e + i$	Use data to offset recirculating index.
	$g \leftarrow$**T**(f)	Load index to array of records.
	$h \leftarrow g$ << 2	Shift index to form offset.
	$i \leftarrow$ **R**.*index* (h)	Load new value of recirculating index.
	$j \leftarrow$ **R**.*value* (h)	Load new data value of interest.
	$k \leftarrow k + j$	Accumulate new data value.
	l: goto loop	Start another iteration.

Figure 3.3. Assembly level representation of simple loop.

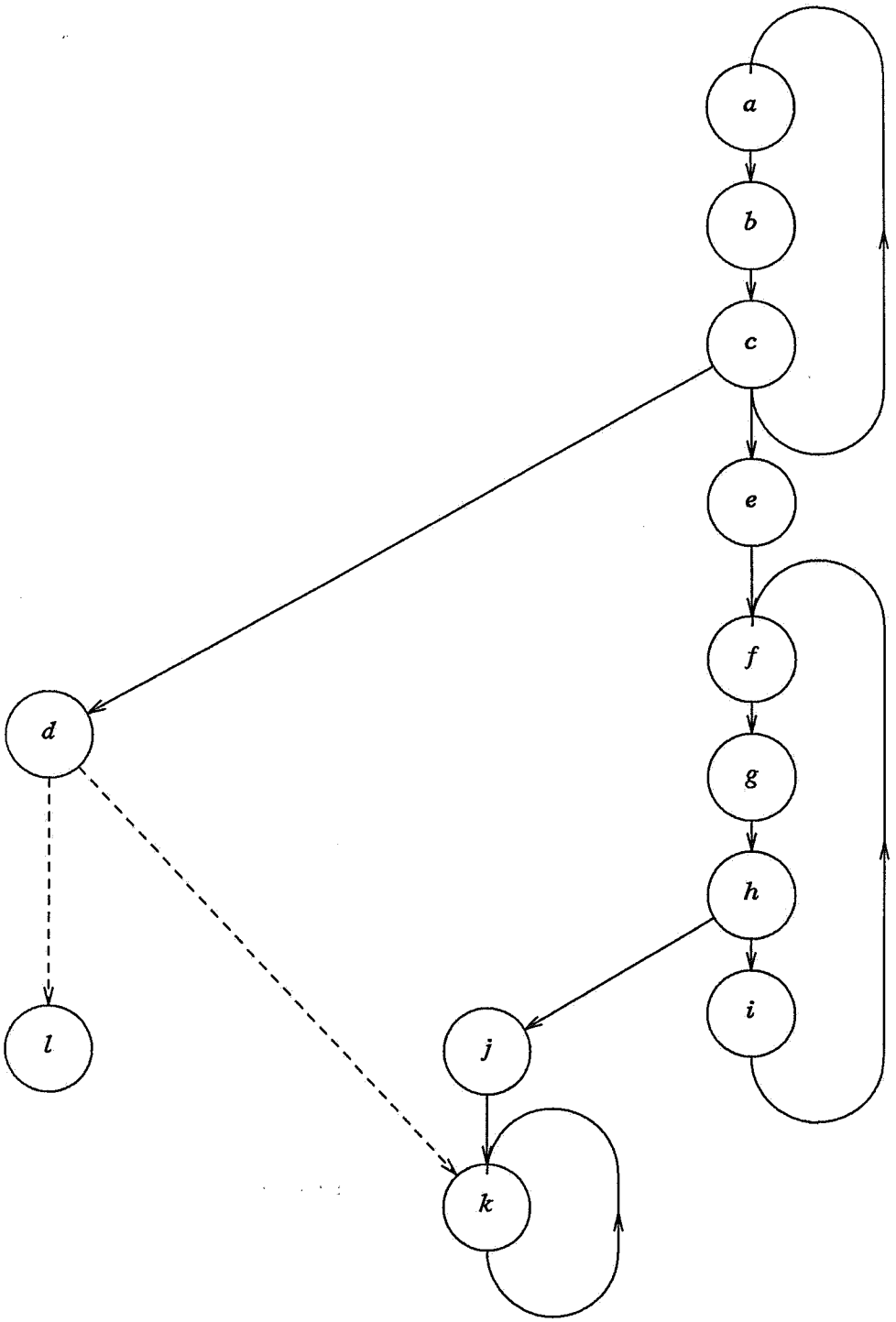

Figure 3.4. Dependency graph for simple loop.

Such a loop, executed on a wide variety of architectures, will achieve varying levels of performance. In this section we compare the advantages and disadvantages of several architectures for this kind of loop.

3.2.1. Scalar Architectures

Consider a pipelined scalar architecture such as the RISC-1 microprocessor[6]. This microprocessor issues one instruction per cycle, and performs a load or delayed branch instruction in two cycles and all other instructions in one cycle. Figure 3.5 shows a table of instruction issue times for one iteration of the loop. Entries in the "INSTRUCTION ISSUED" column contains either the instruction being issued, or ϕ if a data dependency prevents the next instruction from being issued in that clock period.

TIME	INSTRUCTION ISSUED
0	$a \leftarrow \text{arc}(c)$
1	ϕ
2	$b \leftarrow 1(a)$
3	ϕ
4	$c \leftarrow \text{node}(b)$
5	ϕ
6	d : if $(c=0)$ *exit*
7	$e \leftarrow \text{data}(c)$
8	ϕ
9	$f \leftarrow e + i$
10	$g \leftarrow \mathbf{T}(f)$
11	l : goto loop
12	$h \leftarrow g << 2$
13	$j \leftarrow \mathbf{R}.value(h)$
14	$i \leftarrow \mathbf{R}.index(h)$
15	$k \leftarrow k + j$

Figure 3.5. Scalar processor schedule for one iteration of the loop.

Note that the schedule shown in figure 3.5 has been optimized to take maximum advantage of instruction overlap. To facilitate instruction overlap further, we have assumed that delayed branches can be delayed by greater than two cycles if desired. This allows branch instruction l to be issued at time 11, which otherwise would have been a ϕ cycle. An unshown parameter within the branch instruction increases the delay and causes the next iteration to begin at time 16.

As figure 3.5 shows, each iteration of the loop takes 16 clock cycles. Therefore the time for a scalar RISC-1 microprocessor to complete n iterations is

$$T_{SCALAR} = 16n$$

We shall use this time as the basis for comparison with other architectures.

3.2.2. Vector Architectures

Higher levels of performance can be achieved by increasing the level of concurrency beyond that offered by a scalar processor. Suppose that the RISC-1 microprocessor is augmented with vector capabilities similar to those of the Cray-1[1]. In order to use vector instructions, a loop must be distributed into a set of simpler loops such that each new loop corresponds to exactly one vector instruction[16]. This transformation is shown in figure 3.6. In this example we optimistically assume that some unspecified hardware is available for loop control, hence we ignore the *if* and *goto* instructions. Only loops 2, 4, and possibly 5 can be vectorized since the other loops each contain more than one instruction. The reason loops 1 and 3 cannot be distributed further is that the statements within them form *recurrences*[27]. It is well known that recurrences involving pointers through memory or indirect references through arrays such as those in loops 1 and 3 cannot be vectorized and must be executed serially. Loop 5 is also a recurrence, but it is a special case in that it is a vector reduction operation

loop 1:	$a_m \leftarrow \text{arc}(c_{m-1})$	Load pointer to arc pointer list.
	$b_m \leftarrow 1(a_m)$	Load pointer to second arc descriptor.
	$c_m \leftarrow \text{node}(b_m)$	Load pointer to destination of arc.
ignored:	d: if $(c_m = 0)$ *exit*	Terminate loop if no more vertices.
loop 2:	$e_m \leftarrow \text{data}(c_m)$	Load data field from vertex descriptor.
loop 3:	$f_m \leftarrow e_m + i_{m-1}$	Use data to offset recirculating index.
	$g_m \leftarrow \text{T}(f_m)$	Load index to array of records.
	$h_m \leftarrow g_m << 2$	Shift index to form offset.
	$i_m \leftarrow \text{R}.index\,(h_m)$	Load new value of recirculating index.
loop 4:	$j_m \leftarrow \text{R}.value\,(h_m)$	Load new data value of interest.
loop 5:	$k_m \leftarrow k_{m-1} + j_m$	Accumulate new data value.
ignored:	l: goto loop	Start another iteration.

Figure 3.6. Example of vectorization by loop distribution.

involving an associative operator. On some vector machines special hardware is available to evaluate such recurrences quickly with a single vector instruction[28].

From figure 3.6 we can derive the best case execution timing for a vector processor with memory access delay of two cycles and arithmetic computation delay of one cycle. Namely, we assume that the vector processor has a sufficient number of processing elements or pipelines so that resource contention is not an issue. However, we do assume that there is only a single scalar processor with an instruction issue rate of one instruction per clock cycle, since that is usually the case for vector architectures. Loops 1 and 3 must be executed on the scalar processor. Loop 1 contains a chain of three loads so its execution time is $6n$ cycles. Loop 3 contains a chain of two loads and two ALU operations so its execution time is also $6n$ cycles. Loops 2 and 4 each contain one load instruction and so can be vectorized. Since all n iterations of a vectorizable loop can,

with sufficient resources, be performed in parallel, loops 2 and 4 take 2 cycles each. We shall be very optimistic and assume that loop 5 can be treated as if it was a pure vector instruction and charge only 1 clock for its execution. Taken together we find that the best execution time on a vector processor is

$$T_{VECTOR} = 6n + 2 + 6n + 2 + 1 = 12n + 5$$

The speedup relative to the scalar processor is given by

$$SP_{VECTOR} = \frac{T_{SCALAR}}{T_{VECTOR}} = \frac{16n}{12n + 5}$$

For a large number of iterations the speedup approaches

$$SP_{VECTOR} \approx 1.333$$

Note that this speedup is very optimistic and does not take into account overhead for loop control.

In spite of the fact that vector architectures offer much more concurrency, the achieved performance is rather poor for this example. The reason is that only a small fraction of the loop is vectorizable due to the extensive number of recurrences and hence most of the execution must be performed in scalar mode. Almost all linked data structures cause recurrences through memory. The use of linked data structures is pervasive in nonnumeric programs as well as certain numerical programs, such as those that operate on dynamic sparse matrices. In view of the high cost of providing vector execution capabilities, the use of a vector architecture is inappropriate for job loads containing significant usage of linked data structures.

3.2.3. Multiprocessor Architectures

A multiprocessor is more flexible than a vector architecture. A schedule showing issue times for the first three iterations on a multiprocessor consisting of three

independent scalar RISC-1 microprocessors is shown in figure 3.7. In this figure the instructions are represented by either the result register name or the explicit label as

TIME	PROCESSOR		
	1	2	3
0	a		
1	ϕ		
2	b		
3	ϕ		
4	c		
5	ϕ		
6	d	a	
7	e	ϕ	
8	ϕ	b	
9	f	ϕ	
10	g	c	
11	l	ϕ	
12	h	d	a
13	j	e	ϕ
14	i	ϕ	b
15	k	f	ϕ
16		g	c
17		l	ϕ
18		h	d
19		j	e
20		i	ϕ
21		k	f
22			g
23			l
24			h
25			j
26			i
27			k

Figure 3.7. Multiprocessor schedule for three iterations of the loop.

shown in figure 3.3. Because recurrences transmit data to future iterations of the loop, the initiation time of each loop iteration is delayed relative to the previous iteration. Techniques for calculating the value of this delay and transformation algorithms for minimizing this delay have been developed[29]. The schedule shown is the best possible in terms of minimizing the inter-iteration delay, which is six in this example. Provided there are enough processors to eliminate resource conflicts, the execution time to complete n iterations on a multiprocessor is

$$T_{MULTI} = 6n + 9$$

Note that instruction k takes only one time unit to complete execution and that all instructions are actually completed by the time k is completed. The speedup relative to a single scalar processor is therefore

$$SP_{MULTI} = \frac{T_{SCALAR}}{T_{MULTI}} = \frac{16n}{6n + 9} \approx 2.667$$

Since multiprocessors are more flexible than vector processors, they can be expected to achieve higher speedup for a wider class of application programs. In this example a multiprocessor architecture was able to do twice as well as a vector architecture. However in calculating the timing for a multiprocessor we have ignored the communication and synchronization time between processors and the time lost due to memory port contention. If a multiprocessor has independent scalar processing units that operate asynchronously from each other (an asynchronous multiprocessor), then some amount of interprocessor communication overhead is inevitable. When asynchronous multiprocessors are used to evaluate recurrences, the achieved performance is very sensitive to interprocessor communication overhead. For example, even a modest interprocessor communication delay of two clock cycles per iteration would lower the speedup from 2.667 to 2.

The asynchronous characteristic of multiprocessors is advantageous in that it provides greater application flexibility and also allows a larger collection of processors to be coupled together, due to less restrictive clocking requirements. When applied to recurrences, however, these advantages cannot easily be realized because recurrences severely restrict the number of processors that can profitably be used. Referring to figure 3.7, we see that the maximum number of processors that can be used efficiently on this example is three, since the fourth iteration cannot begin until time 18, whereas processor 1 is free by time 16. Thus asynchronously coupling large numbers of processors not only incurs the additional cost of asynchronous interprocessor communication, but provides no performance gain in many applications. If only few processors can be used effectively, a small synchronous system is more cost-effective.

3.2.4. Horizontal Architectures

An alternative to the asynchronous multiprocessor architecture is the *horizontal architecture*[9, 30, 31]. A horizontal architecture consists of multiple processing elements controlled by a single instruction issue unit. These architectures use wide instructions with multiple fields to control each processing element independently in a manner similar to horizontal microcode. The processing elements may be specialized pipelined functional units[30, 31] or relatively unspecialized scalar processors[9]. The main advantage of horizontal architectures is that they are globally synchronized and extremely tightly coupled. These characteristics allow horizontal architectures to provide low-overhead high-bandwidth communication between processing elements at relatively low cost. The disadvantage is that relatively fewer processing elements can be supported by such architectures. This, though, is not a serious handicap for recurrence intensive workloads which cannot effectively utilize many processors in any case.

Consider a horizontal architecture that consists of a two-segment pipelined memory reference unit, a one-segment ALU, and a two-segment pipelined delayed jump unit. This configuration was chosen to be compatible with the RISC-1 microprocessor being used as the basis for comparison. The separation of functions is motivated by the observation that distinct specialized hardware is needed for each of the pipelines, hence parallel execution of these functions is reasonable.

Figure 3.8 shows the execution schedule for one iteration of the loop on this processor. In this figure ϕ cycles are shown as blank entries. Note that the time for one iteration is 15 cycles instead of 16 cycles for the RISC-1 microprocessor. This reduction arises because the increased concurrency allows instructions d and e to be issued in the same clock cycle. However, if only this small speedup of 1.067 were attained it would

TIME	MEM		ALU	JMP	
	seg. 1	seg. 2	seg. 1	seg. 1	seg. 2
0	*a*				
1		*a*			
2	*b*				
3		*b*			
4	*c*				
5		*c*			
6	*e*			*d*	
7		*e*			*d*
8			*f*		
9	*g*				
10		*g*			
11			*h*		
12	*j*				
13	*i*	*j*		*l*	
14		*i*	*k*		*l*

Figure 3.8. Horizontal architecture schedule for one iteration of the loop.

hardly be worthwhile in view of the higher cost of a horizontal architecture.

High performance is achieved on a horizontal architecture when the execution of multiple loop iterations is overlapped. The left hand side of figure 3.9 shows one iteration of the schedule for the loop. Several extra delays have been added to the compact schedule shown in figure 3.8 for reasons that will become apparent. Note that only the first segment of the MEM and JMP pipelines have been shown since the behavior of the second segment can be inferred from the first segment.

The right hand side of figure 3.9 shows a composite schedule obtained by overlapping four copies of the left hand side schedule each delayed by seven clocks. This time delay between iterations is called the *initiation interval.* The superscript indicates the iteration number associated with each particular instruction. An initiation interval of seven clocks is sufficient to satisfy dependencies between iterations. The inter-iteration dependencies are $c^m \rightarrow a^{m+1}$, $i^m \rightarrow f^{m+1}$, and $k^m \rightarrow k^{m+1}$.

As shown in the figure, beginning with clock 14 the memory pipeline is fully utilized and continues to be fully utilized throughout future iterations. Therefore an initiation interval of 7 is minimum. The execution time on the horizontal architecture is

$$T_{HORIZONTAL} = 7n + 16$$

Hence the speedup relative to a scalar processor is

$$SP_{HORIZONTAL} = \frac{T_{SCALAR}}{T_{HORIZONTAL}} = \frac{16n}{7n + 16} \approx 2.286$$

While this speedup not as high as the asynchronous multiprocessor speedup of 2.667, it should be noted that communication between processing elements in a horizontal architecture does not introduce any overhead since interprocessor synchronization is precalculated at compile time. The multiprocessor required three memory ports and three ALUs to achieve a speedup of 2.667 while the horizontal architecture required only one

TIME	MEM	ALU	JMP
0	a		
1			
2	b		
3			
4	c		
5			
6			
7			d
8			
9			
10	e		
11			
12			
13			
14		f	
15	g		l
16			
17		h	
18			
19	i		
20	j		
21			
22		k	

TIME	MEM	ALU	JMP
0	a^1		
1			
2	b^1		
3			
4	c^1		
5			
6			
7	a^2		d^1
8			
9	b^2		
10	e^1		
11	c^2		
12			
13			
14	a^3	f^1	d^2
15	g^1		l^1
16	b^3		
17	e^2	h^1	
18	c^3		
19	i^1		
20	j^1		
21	a^4	f^2	d^3
22	g^2	k^1	l^2
23	b^4		
24	e^3	h^2	
25	c^4		
26	i^2		
27	j^2		

Figure 3.9. Horizontal architecture schedule showing iteration overlap.

memory port and one ALU to achieve almost as high a speedup. Therefore the horizontal architecture can be expected to be less costly than the multiprocessor. Furthermore, considering that the asynchronous multiprocessor speedup drops to 2.000 with the introduction of even very modest communication delay, the horizontal architecture can be expected to outperform the asynchronous multiprocessor for many job loads.

3.2.5. Summary

High performance computer systems need to be efficient both on vectorizable numerical job loads as well as on other more general job loads. The extensive use of linked data structures causes many recurrences through memory. These recurrences cannot be transformed into vector form, hence vector architectures cannot provide substantial speedup on such job loads. Recurrences also limit the number of processors that can be used profitably in a multiprocessor architecture. This, coupled with the fact that asynchronous multiprocessors generally have nonnegligible interprocessor communication overhead, suggests that asynchronous multiprocessors consisting of a number of conventional scalar processors may not be the most cost-effective architecture for general job loads.

The use of horizontal architectures appears to offer improved performance over vector architectures, and improved cost-effectiveness over asynchronous multiprocessors. High utilization of horizontal multiprocessors can be obtained by code scheduling so as to maximize overlap between loop iterations. In the next sections we develop a technique for the automatic generation of optimal throughput loop schedules for horizontal architectures.

3.3. Scheduling Graphs with Acyclic Dependencies

The complexity of finding an optimal throughput schedule for a loop is highly dependent on the characteristics the dependency graph representing the loop. For purposes of code scheduling it is convenient to classify the nodes in a dependency graph, **N**, into subsets

$$\mathbf{N} = \bigcup_{k=1}^{m} \mathbf{R}_k \bigcup \mathbf{S} \bigcup \mathbf{A}$$

This classification is based on *strongly connected components*[32]. Strongly connected components of a directed graph are defined to be maximal sets of nodes such that if nodes x and y are members of the same strongly connected component, then there is a directed path from x to y and also a directed path from y to x. Each strongly connected subgraph containing two or more nodes is called a *multinode recurrence*, denoted by the set $\mathbf{R}_k$. The total number of such multinode recurrences is m. Other strongly connected subgraphs contain only one node each and are called *self-loops*, denoted by the set **S**. The remaining nodes are not in any strongly connected components. These nodes are classified as *acyclic*, denoted by set **A**. Note that the node sets $\mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_m$, **S** and **A** are all disjoint.

The problem of optimal throughput loop scheduling is to find a valid schedule with a *minimum initiation interval* (*MII*). The simplest case occurs when the dependency graph is acyclic, e.g. the graph contains only acyclic nodes. An optimal scheduling technique for acyclic dependency graphs has been proposed by Rau[31] for use on horizontal architectures. In this section we briefly review Rau's results.

3.3.1. The Available Resource Limit Constraint

If a graph contains only acyclic nodes then every iteration of the loop is independent of every other iteration. Consider the acyclic dependency graph with nodes **N** = **A**

shown in figure 3.10. This graph was derived from figure 3.4 by ignoring the feedback arcs, thus removing inter-iteration dependencies. The letters M, A, and J within each node specify the pipelined functional unit (MEM, ALU, and JMP, respectively) required by that node. The number in each node, of the form $+s$, gives the number of segments in the functional unit pipeline.

Since there are seven memory references needed per iteration, and only one pipelined memory unit is available, successive iterations cannot be initiated less than seven clocks apart. In general there is a lower bound on the *MII* based on resource constraints. This lower bound is called the *available resource limit* (*ARL*) and is defined as follows.

$$ARL(\mathbf{N}) = \max_{c \in \mathbf{C}} \sum_{i=1}^{n} \delta_i(c)$$

Where

$$\delta_i(c) = \begin{cases} 1 & \text{if } f_i = c \\ 0 & \text{otherwise} \end{cases}$$

Nodes are numbered from 1 to n, indexed by i. The *type* of functional unit required by a node is given by f_i. **C** denotes the set of all functional unit types. In this example, $\mathbf{C} = \{\mathrm{M}, \mathrm{A}, \mathrm{J}\}$.

The available resource limit states that the initiation interval is lower bounded by the most heavily used resource. Since, for this example, $\sum_{i=1}^{n} \delta_i(\mathrm{M}) = 7$, $\sum_{i=1}^{n} \delta_i(\mathrm{A}) = 3$, and $\sum_{i=1}^{n} \delta_i(\mathrm{J}) = 2$, the *ARL* is 7. This lower bound is not restricted to schedules for acyclic dependency graphs. In general, a valid schedule for any dependency graph must satisfy the condition

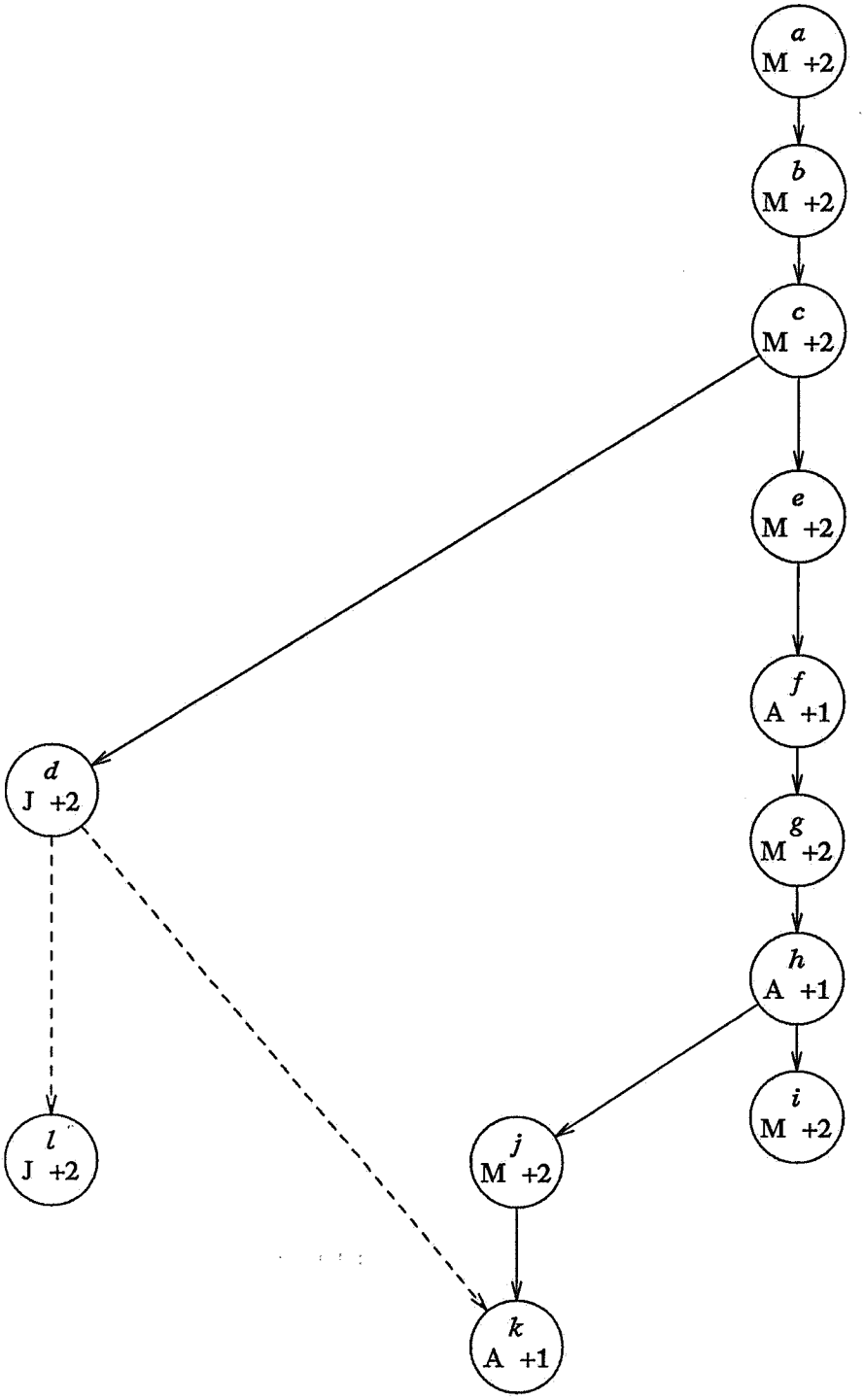

Figure 3.10. Example of acyclic data dependency graph.

$$MII(\mathbf{N}) \geq ARL(\mathbf{N})$$

where N is the set of all nodes in the graph.

Patel[33] has shown that for strictly acyclic dependency graphs, at least one functional unit can always be fully saturated. Therefore

$$MII(\mathbf{A}) = ARL(\mathbf{A})$$

Once the *MII* has been found, it is relatively easy to construct a valid schedule with that initiation interval.

Before presenting the algorithm for schedule construction, some terminology is necessary. A *schedule* for a set of nodes $\mathbf{N} = \{1, \ldots, n\}$ is given by the issue times of each node, denoted by x_i, $1 \leq i \leq n$. The initiation interval for a schedule is denoted by p. The execution delay of a node is equal to the number of segments in the pipelined functional unit used by that node. This delay is given by s_{f_i}, where f_i is the functional unit used by node i.

A *resource conflict* occurs if two nodes require the same functional unit in the same clock cycle. Note that since successive iterations of a loop are overlapped with an initiation interval of p, a node from iteration k issuing at time t will conflict with a node from iteration $k-1$ issuing at time $t+p$, provided both nodes require the same functional unit. More generally, the *modulo usage function* is defined as

$$\delta_i(c, t) = \begin{cases} 1 & \text{if } f_i = c \text{ and } (x_i \bmod p) = t \\ 0 & \text{otherwise} \end{cases}$$

This function is 1 if and only if a node i requires resource c at any of the times $t, t+p, t+2p, \cdots$ For a schedule to be valid (e.g. causes no resource conflicts), the following condition must hold.

$$\sum_{i=1}^{n} \delta_i(c, t) \leqslant 1 \qquad \textit{for all } c \in \mathbf{C},\ 0 \leqslant t < p$$

An algorithm for finding an optimal schedule for an acyclic graph is given in figure 3.11. Note that nodes in a directed acyclic graph can always be topologically sorted to get a linear sequence in which every node depends only on nodes preceding it in the sequence. Without loss of generality we can assume that the set of nodes N is so ordered. An informal description of algorithm A follows:

(1) The first node is not dependent on any other node, hence step A4 does nothing. Step A5 also does nothing since no resource has been reserved for any node. Therefore $x_1 = 0$.

(2) Suppose that a partial schedule consisting of nodes 1 through $j-1$ has already been found. The earliest issue time for node j can be computed by examining all the predecessor nodes that j depends on, given by the set $pred(j)$. This examination is carried out in step A4. Since the nodes are being processed in topological order, all the predecessors of j must have already been processed and hence their x_i are well defined.

A1. $p \leftarrow ARL(\mathbf{N})$

A2. **for** $j \leftarrow 1$ **to** n **do**

A3. $\quad x_j \leftarrow 0$

A4. $\quad$ **for** *each* $i \in pred(j)$ **do** $x_j \leftarrow \max\{ x_j, x_i + s_{f_i} \}$

A5. $\quad$ **while** $\sum_{i=1}^{j-1} \delta_i(f_j, x_j \bmod p) \neq 0$ **do** $x_j \leftarrow x_j + 1$

Figure 3.11. Algorithm A: optimal throughput schedule for acyclic graphs.

---

(3) If issuing node j at the earliest permissible time does not cause a resource conflict then node j is assigned to that time. Otherwise the starting time of node j is incremented until there is no conflict. This resolution of resource conflicts is carried out in step A5. Note that because p has been calculated to be just large enough so as not to overutilize any resource, the while-loop in step A5 terminates after at most $p-1$ iterations.

The application of algorithm A to the graph of figure 3.10 yields the schedule shown in figure 3.12.

Because every iteration of the loop is identically scheduled, A complete characterization of the steady state behavior of the loop can be obtained by looking at p consecutive clock cycles. One can simply divide a schedule into sections each p clock cycles long, label successive sections with successively decremented iteration superscripts, and overlay them on top of one another. This overlaid representation is called a *modulo reservation table* (MRT). Figure 3.13 shows the MRT which corresponds to the example schedule. The superscript gives the iteration numbers. The subscript gives the issue time of the node in clock cycles relative to the beginning of the schedule. Note that the issue times are redundant; a node z^{-k} is issued at time $x_z = t + kp$, where t is the *slot time* in the MRT. For example, the node j^{-2} has a slot time of 5, hence its issue time is $5 + 2 \cdot 7 = 19$. Although redundant in this example, the issue time is included to be compatible with later examples.

In addition to being a more compact representation of the schedule, the MRT is also a convenient data structure for the evaluation of the while condition in step A5 of algorithm A. The while-loop terminates when a time slot is found whose use by the current node will not cause resource conflicts. By using the MRT, the while-loop terminates when there is an empty slot in the appropriate functional unit column. A

TIME	MEM	ALU	JMP
0	*a*		
1			
2	*b*		
3			
4	*c*		
5			
6	*e*		*d*
7			
8		*f*	*l*
9			
10	*g*		
11			
12		*h*	
13			
14			
15	*i*		
16			
17			
18			
19	*j*		
20			
21		*k*	

Figure 3.12. Schedule for acyclic dependency graph.

resource conflict occurs if two nodes are assigned to the same slot in the MRT. Full utilization of a resource occurs when a column is completely filled. In this example the MEM column is full, hence the memory pipeline is fully utilized and optimal throughput that matches $\overset{\cdot}{ARL}$ is obtained.

3.3.2. Startup Time and Scheduling Complexity

Throughput optimal schedules achieve optimal performance only after several loop iterations; the first few iterations are not optimal because resources reserved for

TIME	MEM	ALU	JMP
0	a_0^0	k_{21}^{-3}	
1	i_{15}^{-2}	f_8^{-1}	l_8^{-1}
2	b_2^0		
3	g_{10}^{-1}		
4	c_4^0		
5	j_{19}^{-2}	h_{12}^{-1}	
6	e_6^0		d_6^0

Figure 3.13. Modulo reservation table for acyclic graph schedule.

nonexistent previous iterations are unused. The number of suboptimal iterations is related to the *length* of the schedule, l, defined as

$$l = \left\lfloor \frac{\max_{i \in N}\{ x_i \}}{p} \right\rfloor + 1$$

Referring to figure 3.13, a schedule with length $l = 4$ has nodes belonging to iterations in the range 0 through $-(l-1) = -3$. During iteration 1, the time slots reserved for nodes labeled as iterations -1, -2, and -3 are unused. In general, during iteration k, $1 \leqslant k < l$, the time slots reserved for nodes belonging to iterations labeled $-k, \ldots, -(l-1)$ are unused. Therefore a schedule of length l achieves optimal performance beginning with iteration l.

In algorithm A the assignment of x_j was made based on the first available slot in the MRT. There is no reason why the first empty slot must be assigned. A shorter

schedule may be achieved by assigning x_j to other empty slots, as shown in figure 3.14. The number of assignments to each x_j that must be searched in order to find the shortest schedule is dependent on the characteristics of the dependency graph. Patel[33] has shown that if the earliest allowable issue time for x_j is t, only the assignments $t, t+1, t+2, \ldots, t+p-1$ need to be considered in finding the shortest schedule. Thus the number of assignments per node is upper bounded by p.

The complexity of an optimal throughput scheduling algorithm is strongly dependent on how the algorithm handles the problem of startup penalties. The complexity of algorithm A can be calculated as follows: There are n nodes to be scheduled sequentially. Each node may, in the worst case, depend on every preceding node. Hence the for-loop in step A4 requires $O(n)$ iterations. The while-loop in step A5 requires $O(p)$ iterations. This gives a total complexity of $O(n^2 + np)$ for the algorithm. If, as is commonly the case, the number of predecessors of a node is bounded by a constant,

TIME	MEM	ALU	JMP
0	a_0^0	h_{14}^{-2}	
1	j_{15}^{-2}		d_8^{-1}
2	e_9^{-1}		
3	i_{17}^{-2}	k_{17}^{-2}	l_{10}^{-1}
4	b_4^0	f_{11}^{-1}	
5	g_{12}^{-1}		
6	c_6^0		

Figure 3.14. Optimal throughput schedule with shorter length.

then the complexity reduces to $O(np)$. Note that for acyclic graphs, $p \leqslant n$ since full utilization of at least one resource is achieved.

Algorithm A has a relatively low complexity of $O(np)$ because it generates an optimal throughput schedule but does not guarantee minimum startup penalty. On the other hand, to find an optimal throughput schedule with minimum startup penalty, it is necessary to consider *all* feasible assignments of x_j at step A5 instead of the first feasible assignment. Since there are $O(p)$ feasible assignments per node, the total complexity of such an algorithm is $O(p^n)$. Patel[33] has demonstrated an efficient branch-and-bound algorithm for finding optimal throughput schedules with minimum startup penalty. In view of the fact that loops usually continue for many iterations and therefore the impact of the startup penalty is amortized over a long period of time, it may not be worthwhile to pay the additional computational cost necessary to generate a minimum length schedule. However, the need for minimum length schedules reappears when more general graphs are considered, to be discussed in section 3.5.

The observation that there are multiple feasible assignments of the x_j leads to the minimum complexity optimal throughput scheduling algorithm by Rau[31], shown in figure 3.15. Algorithm B has a lower complexity than algorithm A by making use of the fact that nodes can be placed anywhere into the MRT so long as they are in the proper functional unit class column. A description of algorithm B follows:

(1) Steps B4-B5 finds the earliest issue time for each node. These steps are the same as steps A3-A4 in algorithm A.

(2) Step B6 performs resource assignment. Each node is assigned a slot time, τ_{f_j}, in the MRT. The counters, τ_c, keep track of the next empty slot time for each resource class, $c \in \mathbf{C}$. Note that the available resource limit assures that $\tau_c < p$ for every resource class c.

B1. $p \leftarrow ARL(\mathrm{N})$

B2. for *each* $c \in \mathrm{C}$ do $\tau_c \leftarrow -1$

B3. for $j \leftarrow 1$ to n do

B4. $\quad x_j \leftarrow 0$

B5. $\quad$ for *each* $i \in pred(j)$ do $x_j \leftarrow \max\{ x_j , x_i + s_{f_i} \}$

B6. $\quad \tau_{f_j} \leftarrow \tau_{f_j} + 1$

B7. $\quad x_j \leftarrow p \left\lceil \dfrac{x_j - \tau_{f_j}}{p} \right\rceil + \tau_{f_j}$

Figure 3.15. Algorithm B: minimum complexity optimal throughput schedule.

(3) Step B7 adjusts the issue time of each node such that it falls on the assigned slot time, τ_{f_i}. This adjustment is made by increasing x_j, if necessary, to satisfy the equation $x_j \ mod \ p = \tau_{f_j}$. Since nodes are assigned in topological order, adding extra delay to the issue time of a node can never violate data-dependency constraints in an acyclic graph.

Application of algorithm B to the graph in figure 3.10 yields the schedule shown in figure 3.16. Optimal throughput is achieved since the MEM column is completely filled. However the length of this schedule is seven, significantly longer than the schedule length of four produced by algorithm A. In general algorithm B can be expected to produces longer schedules than algorithm A. This longer length has an adverse impact on loop startup, but may be negligible if the loop continues for many iterations.

The complexity of algorithm B can be calculated as follows: The loop in step B2 requires c iterations. The loop in step B3 requires n iterations. The inner loop in step

TIME	MEM	ALU	JMP
0	a_0^0	f_{28}^{-4}	d_{21}^{-3}
1	b_8^{-1}	h_{36}^{-5}	l_{29}^{-4}
2	c_{16}^{-2}	k_{44}^{-6}	
3	e_{24}^{-3}		
4	g_{32}^{-4}		
5	i_{40}^{-5}		
6	j_{41}^{-5}		

Figure 3.16. Optimal throughput schedule from algorithm B.

B5 requires n iterations in the worst case. Therefore the total complexity of the algorithm is $O(c + n^2)$. Typically, the number of functional unit classes is fixed and the node fan-in is bounded by a constant. Under this assumption c is a constant and the nested loop in step B5 requires a total of $O(n)$ iterations. Thus the complexity of algorithm B can be reduced to $O(n)$. This is clearly the minimal order of complexity since each node must be examined at least once in order to generate code.

3.3.3. Summary

Loops whose dependency graphs are acyclic can be scheduled to achieve optimal throughput using algorithm B. The $O(n)$ complexity of algorithm B is minimal. This result forms the basis for the proposed simple loop scheduling technique to be proposed.

Algorithm B generates schedules that are suboptimal in terms of length. To generate schedules with minimum length requires an algorithm whose complexity is

$O(p^n)$. This high complexity makes the generation of such schedules unattractive for acyclic graphs since the length of a schedule affects only the loop startup penalty and not the loop steady-state performance. However the generation of minimum length schedules is necessary for the more general graphs in section 3.5, which involve multinode recurrences.

3.4. Scheduling Graphs with Self-Loop Dependencies

In the previous section we presented scheduling algorithms for strictly acyclic dependency graphs with nodes $\mathbf{N} = \mathbf{A}$. In this section we present an extension for handling dependency graphs that include self-looping nodes, but no multinode cycles. These graphs have nodes $\mathbf{N} = \mathbf{S} \bigcup \mathbf{A}$.

One common source of self-loops is loop induction variables[34]. An induction variable x is a variable whose only assignments within the loop are of the form $x = x + c$, where c is a constant or loop-invariant value. Optimizing compilers frequently generate induction variables to step through arrays. For example, the Fortran loop in figure 3.17(a) will be transformed into the loop in figure 3.17(b) by an optimizing compiler in order to eliminate the multiplication otherwise needed to evaluate the address of A(i,j). Other sources of self-loops include reduction operators such as vector summation, and traversals of linked data structures. Statements such as $p = p \rightarrow next$ in C or $i = A(i)$ in Fortran are commonly used to traverse linked data structures. Such statements cause self-loops in the dependency graph.

Consider a self-loop node i of the form $x = x + 1$. The operation in this node is dependent on the value generated by the same node in the previous iteration. Since the execution time of node i is given by s_{f_i}, successive iterations of a loop must be separated by at least s_{f_i} clock cycles in order to allow enough time for the addition function in the previous iteration to be completed. This constraint on the *MII* of a

```
dimension A(10,20)
do i = 1, m
        do j = 1, n
                s = s + A(i,j)
```

(a) source

```
do i = 1, m
        a = &A(i,1)                 &A(i, 1) denotes address of element
        do j = 1, n
                s = s + *a          *a denotes load using a as a pointer
                a = a + 10          Note Fortran arrays are column major
```

(b) object

Figure 3.17. Example of induction variable generation.

schedule is given by the *self-loop limit* (*SLL*), defined as

$$SLL(\mathbf{S}) = \max_{i \in \mathbf{S}} \{ s_{f_i} \}$$

For a general data dependency graph the following condition must hold.

$$MII(\mathbf{N}) \geq \max\{ ARL(\mathbf{N}), SLL(\mathbf{S}) \}$$

The existence of self-loops does not change algorithm B previously given for finding optimal throughput schedules, except that p is precalculated in step B1 to be

$$p = \max\{ ARL(\mathbf{N}), SLL(\mathbf{S}) \}$$

Note that since p can be greater than ARL, full utilization of at least one resource can no longer guaranteed. However the resulting schedules are still throughput optimal, since SLL cannot be violated.

It should be noted that many self-loops form *linear recurrences*[27]. Techniques are available for transforming these recurrences into faster forms when the *SLL* constraint prohibits maximum resource utilization. The *SLL* constraint can be relaxed when such transforms are applicable and used. These transformation techniques are compatible with the scheduling techniques proposed in this chapter, but we shall not pursue them further in this thesis.

3.5. Scheduling Graphs with General Dependencies

A general dependency graph may contain acyclic nodes and self-loop nodes, as well as one or more multinode recurrences. Consider a loop containing a multinode recurrence as shown in figure 3.18. Because the back arcs go from the very bottom of the graph to the very top, there can be no overlap between successive iterations. Therefore maximizing the throughput of such a loop is equivalent to minimizing the delay through the acyclic subgraph which excludes the back arcs. The problem of scheduling a set of dependent tasks on a machine with limited resources so that the total delay is minimized is the same problem as the minimum startup penalty scheduling problem, and is known to be NP-hard[29]. However heuristics which work well in general are available in the literature[29, 33]. It should be noted that although obtaining an optimal schedule for multinode recurrences is NP-hard, it is quite practical to do so if the number of nodes involved is small.

Having established that in general obtaining a maximum throughput schedule for a loop containing a single multinode recurrence is NP-hard, one might consider finding a way to decompose a loop containing multiple multinode recurrences into several smaller NP-hard problems, one for each recurrence. Unfortunately this is not possible in general. Consider the example graph in figure 3.4. This graph contains two multinode recurrences $\{a, b, c\}$ and $\{f, g, h, i\}$. Each of these can be individually scheduled

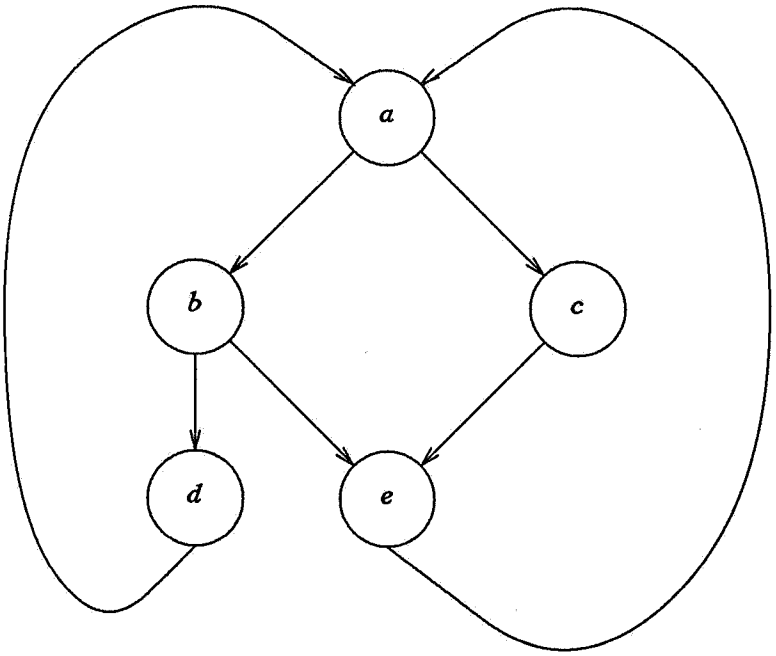

Figure 3.18. Example of multinode recurrence.

with an initiation interval of six cycles as shown in figure 3.19. Since the sum of the resource requirements for both schedules is only five, it would seem possible to combine the two schedules and retain the six cycle initiation interval. However, such a combination cannot be achieved in this case because g and i must be three clocks apart and this separation is incompatible with the two clock separation required between a, b, and c as well as between c and the following a. Since both schedules are rigid in the sense that no node of either schedule can be delayed without increasing the p of that schedule, the two schedules cannot be combined to form a joint schedule with an initiation interval of six.

TIME	MEM	ALU
0	*a*	
1		
2	*b*	
3		
4	*c*	
5		

TIME	MEM	ALU
0		*f*
1	*g*	
2		
3		*h*
4	*i*	
5		

Figure 3.19. Example of separately scheduled multinode recurrences.

Because multinode recurrences cannot in general be decomposed, it is necessary to use a combinatorial technique, such as branch-and-bound, to find the maximum throughput schedule for a set of multinode recurrences. It follows that for a set of multinode recurrences $\mathbf{R} = \{ \mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_m \}$,

$$MII(\mathbf{R}) \geq \max\{ MII(\mathbf{R}_1), MII(\mathbf{R}_2), ..., MII(\mathbf{R}_m) \}$$

For a loop with a general dependency graph whose node set is

$$\mathbf{N} = \bigcup_{k=1}^{m} \mathbf{R}_k \cup \mathbf{S} \cup \mathbf{A}$$

the optimal throughput schedule must also satisfy the available resource limit and the self-loop limit. Therefore, once $MII(\mathbf{R})$ is found by some combinatorial technique, the minimum initiation interval for the entire loop must satisfy

$$MII(\mathbf{N}) \geq \max\{ ARL(\mathbf{N}), SLL(\mathbf{S}), MII(\mathbf{R}) \}$$

At this point the following question arises: Is it possible to construct a schedule for N such that the achieved initiation interval, p, is $\max\{ ARL(\mathbf{N}), SLL(\mathbf{S}), MII(\mathbf{R}) \}$? An affirmative answer to this question would be significant in that once $MII(\mathbf{R})$ is known,

the optimal initiation interval p is readily found. A schedule for that initiation interval must exist, and that schedule must yield the highest steady-state throughput (neglecting startup time).

We have developed an efficient algorithm to construct throughput-optimal schedules for simple loops with general dependency graphs, with the achieved initiation interval $p = \max\{ ARL(\mathbf{N}), SLL(\mathbf{S}), MII(\mathbf{R}) \}$. Before presenting the simple loop scheduling algorithm, we first present a formal specification of the optimization problem that must be solved to construct a throughput-optimal schedule.

3.5.1. Formulation of the Optimization Problem

A program loop is represented by a directed data dependency graph $\mathbf{G} = (\mathbf{N}, \mathbf{D})$ containing instruction nodes $\mathbf{N} = \{1, 2, ..., n\}$. Dependencies or edges between instruction nodes are represented by a $n \times n$ dependency matrix $\mathbf{D} = [d_{i,j}]$. If two nodes i and j are independent then $d_{i,j} = \infty$. Otherwise $d_{i,j}$ gives the distance in loop iterations between the source and destination of the dependency. If, in the current iteration, node j is dependent on node i also of the current iteration, then the dependency distance is zero iterations so $d_{i,j} = 0$. If node j is dependent on node i of the previous iteration then $d_{i,j} = 1$. If, through subscript or pointer analysis it is known that node j can only depend on node i of the k^{th} previous iteration, then $d_{i,j} = k$. Note that $d_{i,j} \geqslant 0$ for all $i \in \mathbf{N}$, $j \in \mathbf{N}$.

Recall that the functional unit class used by an instruction node i is given by f_i and so the execution delay of that node is given by s_{f_i}. Previously we have assumed that there is exactly one resource unit of each particular class. We now generalize that assumption by using u_c to denote the number of functional units of a particular class $c, c \in \mathbf{C}$.

Let x_i be the issue time for node i and let the initiation interval for the schedule be p. The problem of constructing a maximum throughput schedule is to assign integer values to x_i such that the initiation interval p is minimized without violating any dependency or resource constraints. The problem of finding a schedule with a minimum initiation interval is formally stated in figure 3.20.

Constraint (1) is necessary to prevent data dependency violations. Suppose j is dependent on i of the same iteration. Then $d_{i,j} = 0$, so the constraint becomes $x_i + s_{f_i} \leqslant x_j$. This inequality simply says that x_i must finish execution before x_j can start. If j is dependent on i from the previous iteration, then there is an extra latitude of p cycles due to the intervening iteration. This latitude is reflected by the term $-p \cdot d_{i,j}$. Note that if j does not depend on i, then $d_{i,j} = \infty$, so constraint (1) becomes vacuous for the node pair (i, j).

Constraint (2) is necessary to preclude resource usage conflicts. Consider a clock cycle t, $0 \leqslant t < p$. Because successive iterations of a loop are overlapped with a shift of p cycles, a node of iteration k scheduled for time t will occur concurrently with a

---

Assign x_i, $i \in \mathbf{N}$, so as to minimize p subject to the constraints

$$x_i + s_{f_i} - p \cdot d_{i,j} \leqslant x_j \qquad i \in \mathbf{N},\ j \in \mathbf{N} \tag{1}$$

$$\sum_{i=1}^{n} \delta_i(c, t) \leqslant u_c \qquad c \in \mathbf{C},\ 0 \leqslant t < p \tag{2}$$

where

$$\delta_i(c, t) = \begin{cases} 1 & \text{if } f_i = c \text{ and } (x_i \bmod p) = t \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.20. Formulation of the optimal scheduling problem.

---

node of iteration $k-1$ scheduled for time $t+p$. In general, all nodes scheduled at t, $t+p$, $t+2p$, $t+3p$, ... will occur simultaneously due to the overlap. Constraint (2) states that the number of resources of a particular class, c, that are needed simultaneously cannot exceed the number of available resources, u_c.

3.5.2. Initiation Interval Extension Theorem

Crucial to the simple loop scheduling algorithm is an efficient method for extending the initiation interval of an existing schedule. This initiation interval extension method is given by the following theorem.

Theorem

Given a valid schedule for a graph N with initiation interval $p' = MII(\mathrm{N})$, a new schedule with initiation interval $p > p'$ can be constructed by reassigning the node issue times such that

$$x_i = x_i' + (p-p')\left\lfloor \frac{x_i'}{p'} \right\rfloor$$

where x_i' denotes the issue time of node i in the old schedule and x_i denotes the starting time in the new schedule. Note that this expansion adds $p-p'$ empty rows to the end of the old MRT for the p' schedule.

Proof:

The new schedule is valid if and only if it violates neither the data-dependency constraint (1) nor the resource utilization constraint (2).

Part (a) — Prove data-dependency constraint satisfied for the new schedule:

The data-dependency constraint states that

$$x_i + s_{f_i} - p \cdot d_{i,j} - x_j \leqslant 0$$

for every ordered pair of nodes, (i, j). Substitution yields

$$\left(x_i' + (p-p')\left\lfloor\frac{x_i'}{p'}\right\rfloor\right) + s_{f_i} - (p-p'+p')\cdot d_{i,j} - \left(x_j' + (p-p')\left\lfloor\frac{x_j'}{p'}\right\rfloor\right) \leqslant 0$$

Rearrangement yields

$$\left(x_i' + s_{f_i} - p'\cdot d_{i,j} - x_j'\right) + (p-p')\left(\left\lfloor\frac{x_i'}{p'}\right\rfloor - d_{i,j} - \left\lfloor\frac{x_j'}{p'}\right\rfloor\right) \leqslant 0$$

Since the old schedule was a valid schedule, the original node starting times must satisfy data-dependency constraints. Hence

$$x_i' + s_{f_i} - p'\cdot d_{i,j} - x_j' \leqslant 0$$

Noting that $(p-p') > 0$, it follows that the data-dependency constraint for the new schedule is satisfied if

$$\left\lfloor\frac{x_i'}{p'}\right\rfloor - d_{i,j} - \left\lfloor\frac{x_j'}{p'}\right\rfloor \leqslant 0 \tag{3}$$

The validity of the old schedule implies that

$$x_i' \leqslant x_j' + p'\cdot d_{i,j} - s_{f_i}$$

Since x_i' appears in a nonnegative term in inequality (3), replacing that term with another term that is no smaller yields a tighter constraint. Thus, if

$$\left\lfloor\frac{x_j' + p'\cdot d_{i,j} - s_{f_i}}{p'}\right\rfloor - d_{i,j} - \left\lfloor\frac{x_j'}{p'}\right\rfloor \leqslant 0$$

is satisfied, then (3) and hence the data-dependency constraint for the new schedule is satisfied. Since $d_{i,j}$ is an integer, it can be canceled to yield

$$\left\lfloor\frac{x_j'}{p'} - \frac{s_{f_i}}{p'}\right\rfloor - \left\lfloor\frac{x_j'}{p'}\right\rfloor \leqslant 0$$

This inequality is satisfied since $s_{f_i} \geqslant 0$. Hence Part (a) is proved.

Part (b) — Prove resource utilization constraint satisfied for the new schedule:

The resource utilization constraint states that

$$\sum_{i=1}^{n} \delta_i(c, t) \leqslant u_c$$

for $0 \leqslant t < p$ and $c \in \mathbf{C}$ where

$$\delta_i(c, t) = \begin{cases} 1 & \text{if } f_i = c \text{ and } (x_i \bmod p) = t \\ 0 & \text{otherwise} \end{cases}$$

Let δ_i' refer to the old schedule and δ_i refer to the new schedule as defined below:

$$\delta_i'(c, t) = \begin{cases} 1 & \text{if } f_i = c \text{ and } (x_i' \bmod p') = t \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_i(c, t) = \begin{cases} 1 & \text{if } f_i = c \text{ and } \left((x_i' + (p - p')\left\lfloor \frac{x_i'}{p'} \right\rfloor) \bmod p\right) = t \\ 0 & \text{otherwise} \end{cases}$$

Then the resource utilization constraint is satisfied if $\delta_i'(c, t) = \delta_i(c, t)$ or equivalently if

$$\left(x_i' + (p - p')\left\lfloor \frac{x_i'}{p'} \right\rfloor\right) \bmod p = x_i' \bmod p'$$

for $0 \leqslant i \leqslant n$. Replacement of the *mod* function gives

$$x_i' + (p - p')\left\lfloor \frac{x_i'}{p'} \right\rfloor - p\left\lfloor \frac{x_i' + (p - p')\left\lfloor \frac{x_i'}{p'} \right\rfloor}{p} \right\rfloor = x_i' - p'\left\lfloor \frac{x_i'}{p'} \right\rfloor$$

Rearrangement and cancellation yields

$$\left\lfloor \frac{x_i' + (p - p')\left\lfloor \frac{x_i'}{p'} \right\rfloor}{p} \right\rfloor = \left\lfloor \frac{x_i'}{p'} \right\rfloor$$

Let $x_i' = \alpha p' + \beta$ where $\alpha \geq 0$, $0 \leq \beta < p'$, and α, β are both integers. Then

$$\left\lfloor \frac{\alpha p' + \beta + (p - p')\left\lfloor \frac{\alpha p' + \beta}{p'} \right\rfloor}{p} \right\rfloor = \left\lfloor \frac{\alpha p' + \beta}{p'} \right\rfloor$$

Using the fact that $\left\lfloor \frac{\alpha p' + \beta}{p'} \right\rfloor = \alpha$, the equation reduces to

$$\left\lfloor \frac{\alpha p' + \beta + (p - p')\alpha}{p} \right\rfloor = \alpha$$

Simplifying yields

$$\left\lfloor \alpha + \frac{\beta}{p} \right\rfloor = \alpha$$

The equality is satisfied since $\beta < p' < p$ and α is an integer. Hence Part (b) is proved.

3.5.3. The Simple Loop Scheduling Algorithm

The ability to extend the initiation interval of an existing schedule to accommodate more nodes allows us to adapt algorithm B to the more general case involving multinode recurrences. Recall that algorithm B requires that the nodes of the acyclic graph be ordered in topological ordering. Topological ordering of the nodes in a cyclic graph is obviously impossible. Instead we use a topological ordering based on the acyclic superstructure graph[32] of the general graph, constructed as follows:

(1) For each multinode recurrence $\mathbf{R}_k$ do the following: Delete all dependency arcs whose source and destination are both in $\mathbf{R}_k$. Replace the nodes in $\mathbf{R}_k$ by a single new node, r_k, and connect all remaining dependency arcs to and from nodes in $\mathbf{R}_k$

to r_k.

(2) Delete all self-looping dependency arcs from the node set **S**. At this point the graph is acyclic.

(3) Topologically order the acyclic graph and let the nodes form a sequence. For each single node, r_k, representing a multinode recurrence, $\mathbf{R}_k$, do the following: Expand each r_k back into the multinode recurrence $\mathbf{R}_k$. Place the nodes in $\mathbf{R}_k$ into the sequence so that if r_k was between the nodes x and y, then all the nodes in $\mathbf{R}_k$ are placed between x and y. Reconnect the dependency arcs at r_k to the appropriate individual nodes in $\mathbf{R}_k$ as before.

(4) Restore the self-looping dependency arcs from the node set **S**. At this point the original graph has been restored.

This procedure produces the node ordering

$$\mathbf{N} = \{ \mathbf{N}_0, \mathbf{R}_1, \mathbf{N}_1, \mathbf{R}_2, \mathbf{N}_2, \cdots, \mathbf{R}_m, \mathbf{N}_m \}$$

with the property that a set of nodes, $\mathbf{N}_k$, contains the nodes that, in the topological ordering of the acyclic superstructure graph, fell between nodes r_k and r_{k+1}. Note that

$$\bigcup_{k=0}^{m} \mathbf{N}_k = \mathbf{S} \bigcup \mathbf{A}$$

The simple loop scheduling (SLS) algorithm, shown in figure 3.21, uses this node ordering. Note that the SLS algorithm finds a maximum steady-state throughput schedule for the loop, but makes no attempt to reduce loop start-up time. A description of the SLS algorithm follows.

(1) Step C2 constructs an optimal throughput schedule $\{ x_1', x_2', \ldots, \}$ with initiation interval p' for the set of multinode recurrences $\mathbf{R} = \{ \mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_m \}$, using a combinatorial search procedure. This schedule is extended, if necessary, in step C3

C1. **procedure** simple_loop_scheduler (**N**)
C2. construct *MII* schedule for **R**
C3. extend schedule such that $p = \max\{ ARL(\mathbf{N}), SLL(\mathbf{S}), MII(\mathbf{R}) \}$
C4. reserve_time_slots (**R**)
C5. assign_issue_times ($\mathbf{N}_0$)
C6. **for** $k \leftarrow 1$ **to** m **do**
C7. delay_issue_times ($\mathbf{R}_k$)
C8. assign_issue_times ($\mathbf{N}_k$)
C9. **end**

C10. **procedure** reserve_time_slots (**R**)
C11. **for** *each* $c \in \mathbf{C}$ **do**
C12. **for** $t \leftarrow 0$ **to** $p-1$ **do** $\Upsilon_{t,c} \leftarrow u_c$
C13. $\tau_c \leftarrow -1$; $\Upsilon_{-1,c} \leftarrow 0$
C14. **for** *each* $i \in \mathbf{R}$ **do** $\Upsilon_{x_i \bmod p, f_i} \leftarrow \Upsilon_{x_i \bmod p, f_i} - 1$
C15. **end**

C16. **procedure** assign_issue_times ($\mathbf{N}_k$)
C17. **for** *each* $j \in \mathbf{N}_k$ **do**
C18. $x_j \leftarrow 0$
C19. **for** *each* $i \in pred(j)$ **do** $x_j \leftarrow \max\{ x_j, x_i + s_{f_i} \}$
C20. **while** $\Upsilon_{\tau_{f_i}, f_i} = 0$ **do** $\tau_{f_i} \leftarrow \tau_{f_i} + 1$
C21. $x_j \leftarrow p \left\lceil \dfrac{x_j - \tau_{f_i}}{p} \right\rceil + \tau_{f_i}$
C22. $\Upsilon_{\tau_{f_i}, f_i} \leftarrow \Upsilon_{\tau_{f_i}, f_i} - 1$
C23. **end**

C24. **procedure** delay_issue_times ($\mathbf{R}_k$)
C25. $d \leftarrow 0$
C26. **for** *each* $j \in \mathbf{R}_k$ **do**
C27. **for** *each* $i \in pred(j) - \mathbf{R}_k$ **do** $d \leftarrow \max\{ d, x_i + s_{f_i} - x_j \}$
C28. **for** *each* $j \in \mathbf{R}_k$ **do** $x_j \leftarrow x_j + p \left\lceil \dfrac{d}{p} \right\rceil$
C29. **end**

Figure 3.21. Algorithm C: simple loop scheduling algorithm.

to accommodate the resource requirements of other nodes in the graph using the formula $x_i = x_i' + (p - p')\left\lfloor \frac{x_i'}{p'} \right\rfloor$ previously presented.

(2) Step C4 calls a procedure to reserve the resource time slots used by the schedule for the multinode recurrences. Steps C11-C13 initialize counters, τ_c, and the two-dimensional array, Υ, which represents the modulo reservation table. Each row in Υ represents a time slot. Each column in Υ represents a class of resource. An entry $\Upsilon_{t,c}$ gives the number of *free* resources of class c at modulo time t. Initially, all entries in Υ are set to the number of available resources of the appropriate class, u_c. In addition, a dummy row, -1, has been added to Υ to simplify the handling of the τ_c counters. Step C14 reserves the resources needed by the multinode schedule by decrementing entries in Υ.

(3) Step C5 calls a procedure to assign the issue times for the first set of nodes not involved in multinode recurrences. This procedure is essentially the same as algorithm B, with the exception that multiple resources of the same class are allowed. Step C18-C19 finds the earliest issue time for node x_j. Step C20 finds a time slot that has a free resource of the appropriate class. Step C21 adjusts the issue time of x_j to fall in that time slot. The resource is reserved in step C22. Note that by selecting p to be no less than ARL, it is guaranteed that there will be enough time slots to accommodate all the nodes.

(4) Step C7 calls a procedure to adjust the issue times of nodes in a multinode recurrence. This adjustment is necessary since the multinode recurrence schedule found in step C2 does not take into account dependencies between nodes in a multinode recurrence and other nodes. Steps C26-C27 find the minimum delay that must be added to the issue time of each node in $\mathbf{R}_k$ to satisfy dependency constraints from previously scheduled nodes. This minimum delay is rounded up to

the next multiple of p and added to the issue time of each node in $\mathbf{R}_k$ by step C28. Note that dependencies between nodes in $\mathbf{R}_k$ are guaranteed to be satisfied if every node in $\mathbf{R}_k$ is delayed by the same amount. Also note that, because of the node ordering, nodes in $\mathbf{R}_k$ can only depend on nodes previously scheduled and other nodes in $\mathbf{R}_k$. Thus the minimum delay calculation in step C27 checks only those predecessors of j that are not members of $\mathbf{R}_k$. These predecessors are given by the set $pred(j)-\mathbf{R}_k$.

(5) Step C8 calls a procedure, previously described, to assign the next set of nodes not involved in multinode recurrences.

3.5.4. Example of Schedule Generation by the SLS Algorithm

The operation of the SLS algorithm (algorithm C) is illustrated below, using the example graph from figure 3.4, with

$$u_{MEM} = u_{ALU} = u_{JMP} = 1$$

Note that the alphabetic node ordering has been chosen to conform to the ordering required by algorithm B. Therefore $\mathbf{N}_0 = \phi$, $\mathbf{R}_1 = \{a, b, c\}$, $\mathbf{N}_1 = \{d, e\}$, $\mathbf{R}_2 = \{f, g, h, i\}$, and $\mathbf{N}_2 = \{j, k, l\}$.

(1) The *MII* schedule for the two multinode recurrences $\mathbf{R}_1$ and $\mathbf{R}_2$ is shown in figure 3.22. Recall that the memory pipeline is two stages long and the ALU pipeline is one stage long. The initiation interval $p = 7$ is already large enough to accommodate *ARL* and *SLL*, hence no extension is necessary.

(2) Since $\mathbf{N}_0$ is empty, the first step is to process $\mathbf{R}_1$. However nodes in $\mathbf{R}_1$ have no predecessor nodes outside of $\mathbf{R}_1$ so the delay $d = 0$.

(3) The next step is to schedule nodes in $\mathbf{N}_1$. There are two nodes, d and e, in $\mathbf{N}_1$ and they are both dependent on node c. Hence the earliest issue time for both d and e

TIME	MEM	ALU	JMP
0	a_0	f_0	
1	g_1		
2	b_2		
3		h_3	
4	c_4		
5	i_5		
6			

Figure 3.22. *MII* schedule for multinode recurrences.

is 6. The highest empty slot in the MEM column is row 3, so e is assigned at time 10. The highest empty slot in the JMP column is row 0, so d is assigned at time 7.

(4) The delay for $\mathbf{R}_2$ is computed next. Node f from $\mathbf{R}_2$ is dependent on node e. Since node e was issued at time 10 and the memory pipeline is two stages long, node f must not be issued until time 12. To avoid causing resource utilization conflicts the delay must be adjusted to the next multiple of p. Therefore the issue time of each node in $\mathbf{R}_2$ must be delayed by 14 as shown in figure 3.23.

(5) Finally, nodes from $\mathbf{N}_2$ are inserted as shown in figure 3.24. In this figure, superscripts have been added to indicate relative iterations. As shown by the superscripts, this example schedule contains four overlapped iterations. This schedule produced by the SLS algorithm is the same as the example shown in figure 3.9.

TIME	MEM	ALU	JMP
0	a_0	f_{14}	d_7
1	g_{15}		
2	b_2		
3	e_{10}	h_{17}	
4	c_4		
5	i_{19}		
6			

Figure 3.23. Schedule after $\mathbf{R}_2$ has been delayed.

TIME	MEM	ALU	JMP
0	a_0^0	f_{14}^{-2}	d_7^{-1}
1	g_{15}^{-2}	k_{22}^{-3}	l_{15}^{-2}
2	b_2^0		
3	e_{10}^{-1}	h_{17}^{-2}	
4	c_4^0		
5	i_{19}^{-2}		
6	j_{20}^{-2}		

Figure 3.24. Complete optimal throughput schedule.

3.5.5. Summary

The running time of the SLS algorithm can be derived as follows: Step C2 uses a combinatorial search method so in the worst case this step requires exponential time. However the combinatorial search operates only on the set $\mathbf{R}$. We shall use the notation

$|\mathbf{R}|$ to denote the size of the set $\mathbf{R}$ and $NP(|\mathbf{R}|)$ to indicate exponential complexity over the set $\mathbf{R}$.

Assuming that the size of the set C is constant, the "reserve_time_slots" procedure has complexity $O(p + |\mathbf{R}|)$. However since $|\mathbf{R}| \leqslant n$, the complexity can be expressed as $O(n) + O(p)$.

The "assign_issue_times" procedure is similar to algorithm B and has complexity $O(|\mathbf{N}_k|)$. Assuming that the number of predecessors per node is bounded by a constant, the "delay_issue_times" procedure has complexity $O(|\mathbf{R}_k|)$. Since $\sum_{k=0}^{m} |\mathbf{N}_k| + \sum_{k=1}^{m} |\mathbf{R}_k| = n$, the total complexity of steps C5-C8 is $O(n)$.

Taken together the complexity of algorithm C is

$$NP(|\mathbf{R}|) + O(n) + O(p)$$

This complexity indicates that our algorithm is very efficient if either (i) the number of nodes in multinode recurrences is small, or (ii) the combinatorial search algorithm for scheduling multinode recurrences is efficient. In conventional job loads, simple loops are usually small; larger loops almost always involve nested conditional statements and hence cannot be processed by SLS. The number of nodes in multinode recurrences in a small loop is of course also small. Therefore the contribution of the $NP(|\mathbf{R}|)$ term to the complexity measure should not preclude algorithm C from being used in practice.

Aside from the NP term, the remaining complexity, $O(n) + O(p)$, is optimal because (i) every node must be visited at least once to generate code, and (ii) every instruction cycle (every row in the MRT) must also be visited at least once. Since there are $O(n)$ nodes and $O(p)$ instruction cycles, the complexity of $O(n) + O(p)$ is clearly minimal.

Significantly, our algorithm is able to generate an optimal throughput schedule in linear time if the loop is vectorizable on a conventional vector machine. If a loop is vectorizable, then there are no multinode recurrences within the loop. Hence $\mathbf{R} = \phi$, so the NP term vanishes in the complexity measure.

CHAPTER 4

MACHINE ORGANIZATION AND CODE GENERATION ISSUES

4.1. Introduction

The study of compiler code generation techniques is necessarily dependent on the choice of target machine model. One of the most important considerations in choosing a target machine model is the level of abstraction. A highly abstract model is advantageous in that the scheduling algorithms developed for such models are unencumbered by implementation details. This may lead to clean theoretical results which give insights to the solution of global problems. Unfortunately some of the implementation details ignored by a highly abstract model may turn out to be critically important constraints or efficiently exploitable architectural features. In this thesis we have chosen to develop scheduling algorithms based on a fairly detailed machine model in order to explore the relationship between machine organization, instruction set architecture, and compiler code scheduling techniques. This chapter describes the target machine model, discusses implementation considerations that motivated the machine organization, and develops solutions to the practical code generation problems of register assignment and branch handling.

The proposed target machine is a Tightly-coupled Heterogeneous Multiprocessor (THUMPER). This type of machine is characterized by the following attributes:

Tightly-coupled

System synchronization is provide by a single system-wide clock to achieve low-overhead interprocessor synchronization. Individual processors are interconnected by a high-bandwidth low-delay network to provide high-speed interprocessor

communication.

Heterogeneous multiprocessor

A high degree of concurrency is provided through multiple processors, each of which may be pipelined to increase performance further. Improved cost-effectiveness is attained through the capability to mix identically replicated general-purpose VLSI processors and heavily pipelined special-purpose functional units, and by the capability to parameterize both the size of the multiprocessor system as well as the composition of the processors.

Figure 4.1 shows an example configuration of a THUMPER with three processors: an integer arithmetic processor, a floating-point arithmetic processor, and a memory access processor. The processors are interconnected by a crossbar network with embedded

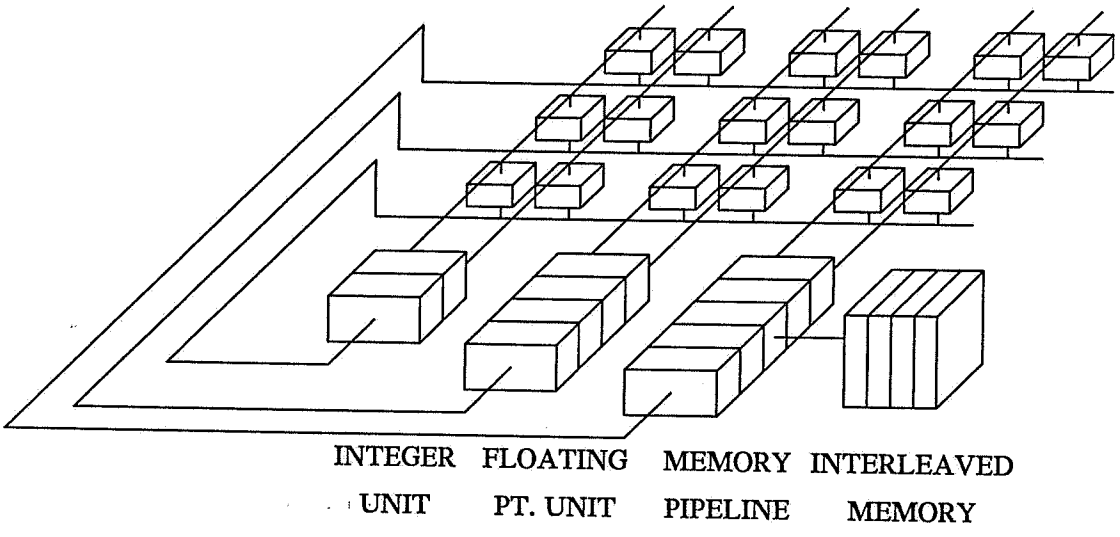

Figure 4.1. Block diagram of a THUMPER configuration.

storage capability at each crosspoint[35]. In the following sections we discuss some of the design considerations that lead to this machine organization.

4.2. Processors and Memory System Design

The design of processors is influenced by two conflicting requirements: low cost and high performance. In a multiprocessor organization, low cost can be achieved by replicating identical multifunction processors, particularly using VLSI technology. On the other hand to achieve very high performance it is usually more cost effective to use a suite of heavily pipelined specialized functional units. A heterogeneous multiprocessor organization captures the advantage of both by incorporating distinct classes of processors. The organization of each class of processors can be optimized to achieve maximum cost-performance. Additional parallelism can be provided through replication.

In our study of code generation techniques we have found that certain constraints on processor designs significantly simplifies and/or improves the running speed of scheduling algorithms. Here we distinguish between explicitly-scheduled resources and implicitly-scheduled resources. An explicitly-scheduled resource is a resource for which contention may occur. An implicitly-scheduled resource is a resource whose availability is guaranteed provided that its associated explicitly-scheduled resource was available at some prior time. For example, the first stage of a linearly pipelined functional unit shared by multiple processors is an explicitly-scheduled resource while all the subsequent stages are implicitly-scheduled resources since the availability of the first stage guarantees the availability of all subsequent stages at the proper later time. From the point of view of code generation, explicitly-scheduled resources are the only ones that need be considered, and we shall use the term *resource* to mean explicitly-scheduled resources unless otherwise noted.

A most important consideration is the number of resources and the time lag between uses of those resources for each individual instruction. All such resources and their relative time slots must be considered when scheduling each instruction. The simplest case is when every instruction uses exactly one (explicit) resource. In this case deciding whether a particular instruction can be scheduled for execution requires examining only a single variable at a single instant in time. If an instruction uses multiple resources during different phases of its execution, then it becomes necessary to examine multiple variables at different times to decide when an instruction can be issued without subsequent conflict with other instructions. The need to check resources at different times causes difficulty during the scheduling of loops because decisions made at the beginning of the loop must be sensitive to conditions at the end of the loop (of the previous iteration) which are yet unknown.

In our target machine we have decided to allow only one explicitly-scheduled resource per instruction and require that each resource be capable of accepting a new instruction per clock. This means that if, for an instruction such as $z = x + y$, the explicitly-scheduled resource is the adder, then the register file containing x, y, and z as well as all the interconnecting busses must all be implicitly-scheduled resources. In other words, dedicated register file ports and busses must be associated with the adder. Furthermore, the adder itself must be a simple pipeline with no shared or looping stages. For multifunctional processors, one resource per instruction implies that every function must take the same amount of time to flow through the pipeline, otherwise the output bus becomes another explicitly-scheduled resource.

Another important consideration is the number of processor classes, and their characteristics, that can execute a particular instruction. From a hardware utilization point of view it may be desirable to provide both a fast floating-point adder as well as

slower microcoded floating-point point addition on a multifunctional processor. Unfortunately to utilize such a system fully the compiler must, for each floating-point add instruction, choose between using the fast adder and possibly delaying another more critical add instruction or using the slower multifunctional processor and possibly delaying several more critical instructions of as yet unknown classes. Choices of this type are called global choices since one decision may impact other choices in the future. In contrast, choosing among several dedicated adders of identical delay is called a local choice since this decision has no effect on choosing a schedule time for other instructions.

To generate high quality code, a compiler must occasionally backtrack and reconsider earlier decisions. However only global choices must be reexamined. Hence, to reduce compilation time, it is desirable to minimize the number of global choices that must be made per instruction. In our target machine we have decided to partition the processors into classes and bind every instruction to a single processor class. Processors of a given class are therefore identical and hence it is a local choice to decide which one to use. With this constraint the only global choice is to decide when to execute a particular instruction.

Determinacy of execution time is another important consideration. Resolution of conflicting resource requirements at compilation time is highly desirable to avoid the cost, in both hardware and run time, needed for arbitration and synchronization. To achieve this resolution, the compiler must be able to predict the exact execution time of every instruction. Because we already restrict processors to be linear pipelines, the execution time of most instructions is completely deterministic. The time for a memory reference, however, cannot be made deterministic because of memory bank conflicts and/or concurrent input/output operations. We have chosen to model the memory sys-

tem as a set of pipelines whose length is equal to the memory reference time in clock cycles. When bank conflicts occur, the entire machine is frozen until the conflicts are resolved.

4.3. Storage-Enhanced Crossbar Interconnect Design

As shown in figure 4.1, the heterogeneous processors and memory pipelines are interconnected by a crossbar with embedded storage at each crosspoint. The storage-enhanced crossbar interconnect was chosen because it simplifies and/or solves a number of difficult implementation and code-scheduling issues. The concept of a storage-enhanced crossbar interconnect is not new, and many of its advantages have been documented[35]. We now briefly review some of these advantages.

In the previous section we discussed the importance of minimizing the number of explicitly-scheduled resources. The crossbar is the only interconnection network that provides dedicated busses for every processor and memory and a dedicated switch for each possible connection. Therefore all the busses and switches are implicitly-scheduled resources. Other interconnection schemes using shared busses and/or switches necessarily introduce additional explicitly-scheduled resources.

A similar line of reasoning leads to the decision to embed the register file within the crossbar interconnect. In order to avoid generation of additional explicitly-scheduled resources, a register file port with no access conflict must be dedicated to each processor port. As the size of a multiprocessor system increases, it becomes impractical to implement a single centralized multiported register file with the required number of conflict-free ports. One solution is to decentralize the register file by distributing the file memory into each crosspoint of a crossbar interconnect.

Referring to the crossbar in figure 4.1, the register read data busses are shown vertically while the register write data busses are shown horizontally. High write

bandwidth is provided by partitioning the register file and associating a distinct partition with the output port of each processor and memory pipeline. In effect, each output port is connected to a separate file memory so that every processor and memory can simultaneously write into the register file without conflict. High read bandwidth is provided by replicating the data within the register file once for each processor and memory read port. This design allows an arbitrarily large storage-enhanced crossbar to be constructed using two-ported random access memories (RAM).

Replicating data to increase register file read bandwidth increases the cost of the system but poses no problem for the compiler since the replications can be made architecturally transparent. Partitioning the register file to increase write bandwidth, however, cannot be made architecturally transparent and hence has a strong impact on code generation. For the machine configuration shown in figure 4.1, the architectural view of the register file is shown in figure 4.2, assuming for this example that the RAM at each crosspoint contains only four words. Each processor can only write into those registers that belong to the partition corresponding to the row in the crossbar connected to that processor's write bus. However, since data is replicated across all register modules in a row, every processor can read registers belonging to all partitions. Although this distributed register file architecture has implementation merits, it does introduce additional code generation problems. We shall return to this issue in section 4.6.

Another reason for embedding the register file within the crossbar interconnect is to mitigate the high cost of the crossbar network in terms of chip count. Here we assume that the physical size of a chip package is determined by the number of pins and not by the amount of logic contained within the chip. Given that a large number of pins are needed at each crosspoint to interconnect the two orthogonal word-wide data busses, the addition of a small RAM to each simple crosspoint switch should not

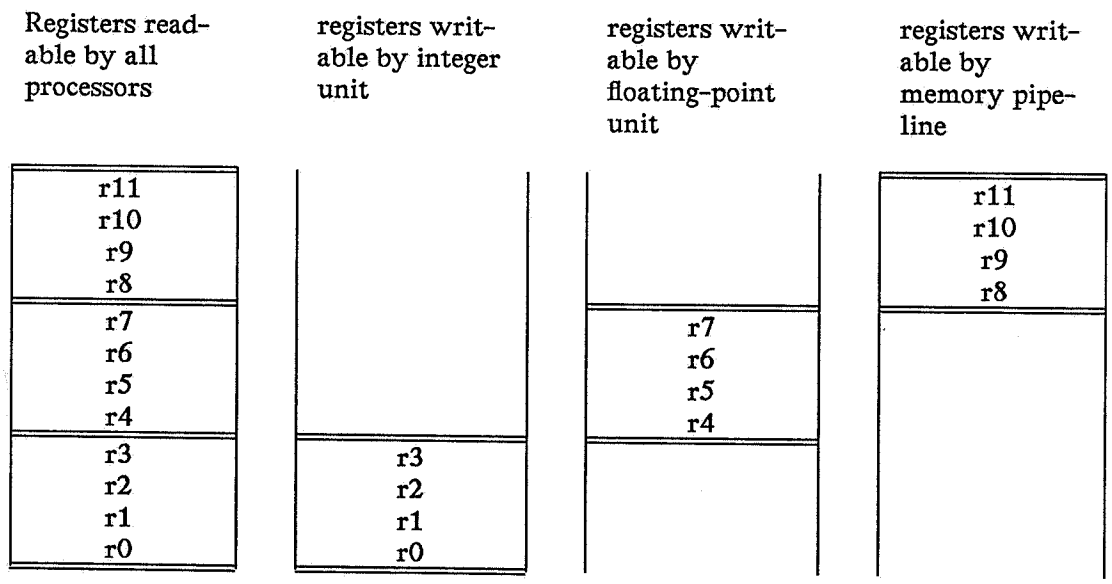

Figure 4.2. Architectural view of register file.

significantly increase the physical chip package size. It therefore appears that storage can be embedded within the crossbar at very low cost.

The scheduling techniques we have developed make use of the substantial local memory within the crossbar interconnect to replace more conventional scalar and vector register files. Moreover our techniques also use these embedded local memories to hold prefetched data, thereby reducing the need for a data cache. In most high performance processors, the physical space devoted to register files and caches is quite substantial[1]. By using the storage-enhanced crossbar interconnect to replace both of these, we feel that the high cost of the crossbar interconnect can be justified in a system context.

4.4. Control Unit Design

The THUMPER is controlled by a single control unit and synchronized by a single system-wide clock. This approach has a number of advantages. Using a centralized clock and a global control unit leads to a highly deterministic system whose detailed run time behavior can be accurately determined at compile time. The compiler can optimize the code by knowing the actual behavior of the machine, instead of knowing only the statistical behavior. Another advantage of using a centralized control unit for a multiprocessor is the elimination of run-time arbitration and synchronization overhead for interprocessor communication.

The THUMPER uses a wide horizontal instruction format as shown in figure 4.3. This instruction format is very similar to that of horizontal microcode, hence architectures based on this type of synchronous multiprocessor organization are also called *horizontal architectures*[9, 30, 31]. A separate field is allocated to each processor, plus an additional field for branch specification or an immediate constant. Each processor field

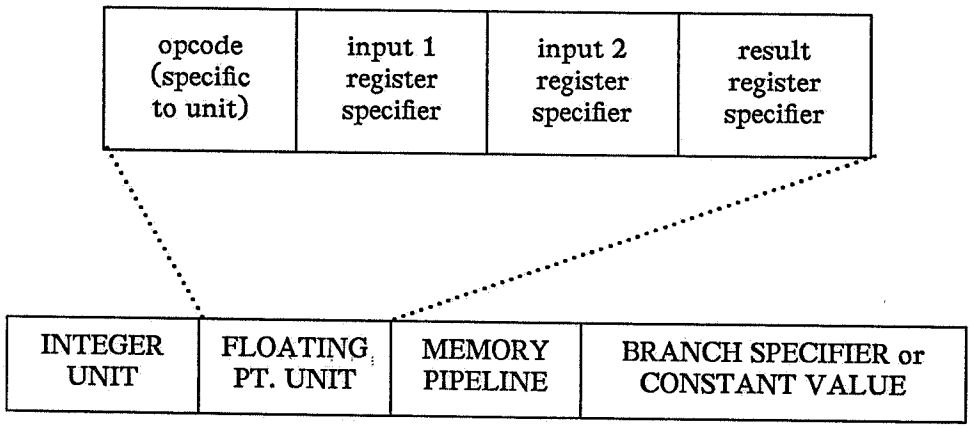

Figure 4.3. Horizontal instruction format.

contains an opcode specific to that processor class. Two register specifier fields are used to address input operands resident in the register file. Another register specifier field is used to address the result operand. Note that the input register specifier fields are large enough to address every partition in the register file while the output register specifier field is large enough to address only the one partition that is writable by that processor.

We chose to use an almost purely horizontal instruction format rather than a more vertical format because we have found that the flexibility offered by horizontal instruction formats is essential for the exploitation of parallelism in a wide range of application programs. As discussed in chapter 3, more highly encoded vertical instruction formats such as those generally employed by vector architectures are unable to exploit much of the parallelism available in those program loops that involve multinode recurrences. Vertical instruction formats are also unsuitable for the exploitation of parallelism available in scalar program fragments.

The control unit is organized as a linear pipeline whose length is equal to the program memory access time plus the time needed to decode the instruction. The program memory is interleaved to supply one instruction per clock cycle. It is further interleaved so that most of the time a branch to an arbitrary bank will experience little or no delay. When a memory bank conflict does occur, the simplest approach is to stop the processor until the conflict is resolved.

We have chosen not to include an instruction cache in our proposed target machine. To be fast, caches must be constructed using a relatively expensive technology, and the physical size of the cache must be kept small in order to reduce cost and minimize the physical separation between the cache and the processor[36]. However, a major reason for incorporating an instruction cache in a machine is to reduce the time needed for a taken conditional branch. Therefore rather than using an instruction

cache, we have chosen to rely on compile-time code scheduling technology to minimize the performance impact of long branch time.

A linear instruction fetch and decode pipeline lends itself naturally to an architecture with delayed branches[5, 6, 7]. Our scheduling techniques are designed to take full advantage of relatively long delay branches, under the assumption that to achieve high clock speed it is necessary to partition the instruction fetch and decode pipeline into multiple segments with fine granularity. The DTS technique performs extensive code rearrangement to allow a sequence of delayed branches to be overlapped, thus reducing the average delay of conditional branches in scalar code.

4.5. Machine Parameters

We have described an expandable multiprocessor organization and discussed the rationale behind some of the design decisions. The significance of this organization is that it can be efficiently implemented using current technology *and* it can be completely characterized by a small number of parameters. The ability to capture concisely all the constraints imposed by the machine organization has a direct impact on the development of scheduling techniques, both in simplifying the algorithms as well as in improving the efficiency of these algorithms. This section describes each of the machine parameters.

The universe of instruction opcodes is given by the set **F**. This set of opcodes defines the functionality of the instruction set architecture. Elements of **F** include the usual integer and floating point arithmetic operations, logical operations, memory operations, etc. The exact membership of **F** is a relatively low-level design issue, and is beyond the scope of this thesis. We do, however, require that **F** include the guarded store and guarded jump instructions described in chapter 2.

The set **C** defines the processor classes. An element of **C** can be a multifunctional processor, such as an integer unit that can perform all the normal arithmetic and logical functions (e.g. an ALU). Elements of **C** can also be unifunctional processors, such as specialized floating point add and multiply pipelines. Since we require a disjoint partition of functionality for different processor classes, a function f_i can be defined to map each instruction opcode i onto one particular processor class.

Multiple processors of the same class can be incorporated for increased parallelism. However we require that all processors belonging to a particular class be functionally identical. The number of replicated processing units of a particular class c is given by u_c.

Since the organization of each processor is constrained to be a linear pipeline and since every instruction is constrained to flow through every pipeline stage, the temporal characteristics of a processor are completely specified by the number of pipeline stages. This number is given by s_c, where c is a processor class. Note that by modeling the memory system and the instruction fetch and decode process as linear pipelines, we can define a "memory" processor class and a "branch" processor class to model the scheduling constraints imposed by the operation of these resources. The parameters for these processor classes are exactly the same as for any other processor class, namely s and u.

Pipelining within the storage-enhanced crossbar interconnect can be handled simply by treating the interconnect pipeline as an extension of the processor pipeline, since the resources within the crossbar are all implicitly-scheduled resources. Therefore the additional delay within the interconnect can be charged to s_c.

The parameters f_i, u_c, and s_c are sufficient for describing the processing part of a THUMPER implementation. The register file is characterized by the size of the RAM within each crosspoint cell. Note that there is no need to specify the number of seg-

ments in the register file since that is implicitly specified by the number of processors, which is equal to $\sum_{c \in C} u_c$.

4.6. Register Assignment Issues

As we alluded to in section 4.3, the distributed register file introduces certain code generation problems that do not arise in a conventional centralized register file. Referring to figure 4.2, the problem with this register file architecture is that to fetch a value it is necessary to know which partition the value is in, i.e. to know which processor generated that value. Sometimes, however, it is impossible to know at compile time which processor will generate a particular value, as shown in the following example.

```
if (...)    x = A[i];
else        x = b * c;
z = x + y;
```

If memory fetches are handled by one processor while arithmetic operations are handled by another processor, the value x must reside in different partitions depending on the outcome of the if-statement. This uncertainty causes problems for the compiler when it tries to generate code for $z = x + y$ since the the location of x cannot in general be determined at compile time. Note, however, that this uncertainty can only occur when the basic block containing $z = x + y$ has two predecessor blocks.

To solve this problem we have elected to constrain the compiler to use registers only for temporaries within a tree of basic blocks. Because each basic block within a tree (except the root) has exactly one predecessor block, it is always possible to identify uniquely the register file partition that a temporary value resides in. During tree transitions all temporary values must be stored in memory. Therefore the fact that there are multiple predecessor blocks branching to the root node of a tree does not cause any problems.

The scheduling technique proposed in chapter 2 directly implements this idea by representing programs as decision trees, hence register assignment with the DTS technique is straightforward. The simple loop scheduling technique proposed in chapter 3 can also implement this idea because an unrolled simple loop forms a highly skewed decision tree. A method of register assignment for the SLS technique is discussed below.

The example optimal throughput schedule produced by the SLS algorithm in figure 3.24 has length $l = 4$. At any one time there are up to four iterations being executed concurrently. Therefore each temporary value name shown in figure 3.24 requires four physical registers to accommodate the four distinct values that exist concurrently. Since each instruction operand specifier must reference four different registers at different times, there is a problem with name binding.

An innovative hardware addressing scheme to solve this name binding problem has been proposed by Rau[35]. This approach uses hardware queues with the capability of deleting any element within the queue. Queues are used at each crosspoint of the crossbar to allow relative register addressing, thus implementing run-time dynamic name binding. Although elegant, the use of hardware queues with random deletion capability rather than RAM to implement the distributed register file significantly increases the complexity of the system and can lead to additional delays in transferring data through the register file. Thus this approach has a negative impact on both performance and cost-effectiveness.

We advocate a much simpler approach that uses additional program storage space to solve the name binding problem statically. Our solution involves unrolling the loop l times, where l is the number of overlapped iterations. Figure 4.4 shows one iteration of the schedule from figure 3.24. Each instruction is shown in detail to illustrate regis-

TIME	MEM	ALU
0	$a_0 \leftarrow \text{arc}(c_3)$	
1		
2	$b_0 \leftarrow 1(a_0)$	
3		
4	$c_0 \leftarrow \text{node}(b_0)$	
5		
6		
7		
8		
9		
10	$e_0 \leftarrow \text{data}(c_0)$	
11		
12		
13		
14		$f_0 \leftarrow e_0 + i_3$
15	$g_0 \leftarrow \text{T}(f_0)$	
16		
17		$h_0 \leftarrow g_0 << 2$
18		
19	$i_0 \leftarrow \text{R}.index\,(h_0)$	
20	$j_0 \leftarrow \text{R}.value\,(h_0)$	
21		
22		$k_0 \leftarrow k_3 + j_0$
23		
24		
25		
26		
27		

Figure 4.4. Detailed representation of a loop schedule.

ter assignments. The JMP processor class instructions have been omited since they are not germane to this discussion.

The register names shown in figure 4.4 have been given subscripts to distinguish among the four physical registers. Note that instructions a, f, and k reference registers with subscript 3, indicating that these values are to come from previous iterations. The schedule shown in figure 4.4 represents the first of the four overlapped iterations. The schedule for the remaining three iterations can be derived by successively

(i) rotating the original schedule by some multiple of p clock cycles, where p is the initiation interval, and

(ii) incrementing the register subscripts by one, modulo l.

Figure 4.5 shows the complete schedule with four overlapped iterations. Note that every instruction now has a unique symbolic register address that can be mapped into a unique physical register address. Therefore the register file can be implemented using an ordinary RAM.

Compared to Rau's dynamic name binding, our approach of loop unrolling to achieve static name binding requires l times more code space per loop. However, we believe that loop unrolling is a more cost-effective solution because it reduces the complexity of the machine. Reduced complexity allows the clock speed to be increased and also reduces the design cost of the machine.

4.7. Architectural Considerations for Delayed Branches

The architecture of branch instructions has a strong impact on the complexity of code optimization techniques. The THUMPER instruction set includes delayed branches with guard expressions as described in chapter 2. This section discusses some architectural considerations for branch instructions.

TIME	MEM	ALU
0	$a_0 \leftarrow \text{arc}(c_3)$	$f_2 \leftarrow e_2 + i_1$
1	$g_2 \leftarrow \text{T}(f_2)$	$k_1 \leftarrow k_0 + j_1$
2	$b_0 \leftarrow 1(a_0)$	
3	$e_3 \leftarrow \text{data}(c_3)$	$h_2 \leftarrow g_2 << 2$
4	$c_0 \leftarrow \text{node}(b_0)$	
5	$i_2 \leftarrow \text{R}.index\,(h_2)$	
6	$j_2 \leftarrow \text{R}.value\,(h_2)$	
7	$a_1 \leftarrow \text{arc}(c_0)$	$f_3 \leftarrow e_3 + i_2$
8	$g_3 \leftarrow \text{T}(f_3)$	$k_2 \leftarrow k_1 + j_2$
9	$b_1 \leftarrow 1(a_1)$	
10	$e_0 \leftarrow \text{data}(c_0)$	$h_3 \leftarrow g_3 << 2$
11	$c_1 \leftarrow \text{node}(b_1)$	
12	$i_3 \leftarrow \text{R}.index\,(h_3)$	
13	$j_3 \leftarrow \text{R}.value\,(h_3)$	
14	$a_2 \leftarrow \text{arc}(c_1)$	$f_0 \leftarrow e_0 + i_3$
15	$g_0 \leftarrow \text{T}(f_0)$	$k_3 \leftarrow k_2 + j_3$
16	$b_2 \leftarrow 1(a_2)$	
17	$e_1 \leftarrow \text{data}(c_1)$	$h_0 \leftarrow g_0 << 2$
18	$c_2 \leftarrow \text{node}(b_2)$	
19	$i_0 \leftarrow \text{R}.index\,(h_0)$	
20	$j_0 \leftarrow \text{R}.value\,(h_0)$	
21	$a_3 \leftarrow \text{arc}(c_2)$	$f_1 \leftarrow e_1 + i_0$
22	$g_1 \leftarrow \text{T}(f_1)$	$k_0 \leftarrow k_3 + j_0$
23	$b_3 \leftarrow 1(a_3)$	
24	$e_2 \leftarrow \text{data}(c_2)$	$h_1 \leftarrow g_1 << 2$
25	$c_3 \leftarrow \text{node}(b_3)$	
26	$i_1 \leftarrow \text{R}.index\,(h_1)$	
27	$j_1 \leftarrow \text{R}.value\,(h_1)$	

Figure 4.5. Complete schedule with register assignments.

Recall that in the description of the decision tree scheduling technique in chapter 2, guarded jumps that branch from an exterior block of a tree to the root of another tree are scheduled in priority order just like any other instruction. The problem with this strategy is as follows. Suppose branches have delay k. This means that the terminal branch on a path through the tree should be scheduled exactly k cycles prior to the end of the path. However, before the path is completely scheduled, the compiler cannot determine how long the path is going to be. Thus until after it has generated the entire path schedule, the compiler cannot determine when the terminal branch instruction should be scheduled.

One possible solution is as follows. Once the entire path schedule has been generated, the compiler can go backward k cycles and insert the terminal branch instruction. Unfortunately there is no guarantee that no other instruction has been scheduled, at the required functional unit, k cycles from the end of the path. In such a case the compiler could insert the branch instruction $k-1$ cycles prior to the end of the path and delay the remaining $k-1$ instructions by one cycle. This solution may be acceptable if the number of instructions that can be issued per cycle is very small, such as one instruction per cycle.

However, for highly concurrent THUMPER configurations that issue many instructions per cycle, this solution is inefficient because no other instruction can be scheduled for the cycle devoted to the inserted branch instruction. Moreover, this solution may introduce inefficiencies into other paths through the decision tree because the active code block at $k-1$ cycles prior to the end of one path may not be the exterior block for that path, but instead may be an interior block shared by several paths. In this case the extra cycle introduced to accommodate the terminal branch instruction for one path causes delays in all other paths that share the interior block into which the

branch instruction is inserted.

Instead, we solve this problem by introducing an *extra-delay* parameter in branch instructions. The extra-delay parameter specifies the number of additional cycles that should be added to the normal delay of the branch. The availability of this extra-delay parameter greatly simplifies the DTS technique *and* eliminates performance degradation due to insertion of terminal branch instructions in highly concurrent THUMPER configurations. With this parameter, the DTS technique simply schedules terminal branch instructions without considering how long the path may be. Once the path has been scheduled, the appropriate extra-delay can be computed and written back into the branch instruction. Naturally, if it turns out that the terminal branch instruction is less than k cycles from the end of the path, then the path must be padded with the appropriate number of no-operation instructions.

The implementation of the extra-delay parameter is straightforward. The branch target address is saved in a register along with the value of the extra-delay parameter. The extra-delay is counted down by hardware, and the branch target address is transferred into the program counter when the extra-delay count becomes zero.

The extra-delay parameter also simplifies the simple loop scheduling technique described in chapter 3. The SLS technique schedules branches strictly based on resource availability and data-dependency constraints, without consideration for the initiation interval. Therefore the loop-completion branch can be scheduled more, or fewer, than k cycles from the end of the modulo reservation table.

If the loop-completion branch is scheduled more than k cycles from the end of the MRT, the extra-delay parameter can be used to increase the branch delay. If the loop-completion branch is scheduled fewer than k cycles from the end of the MRT, the solution is to concatenate one or more copies of the schedule until the loop-completion

branch is no fewer than k cycles from the end of the MRT, and then use the extra-delay parameter as appropriate. Note that multiple copies of a complete schedule such as the one shown in figure 4.5 can be concatenated without change. Therefore, once the SLS algorithm has found a valid schedule for a loop, the branch issues can be quickly resolved.

CHAPTER 5

CONCLUSIONS

5.1. Summary of Results

We have shown that the performance of scalar code can be improved through the use of an integrated design philosophy in which the machine organization, instruction set architecture, and compiler code generation techniques are developed simultaneously. By concentrating our research efforts on the general nature of scalar code, we have insured that our techniques are applicable to a wide range of applications, including job loads that are dominated by nonnumerical and symbolic computations. The results of this research suggests that cost-effective techniques can be used to achieve significant speedup in the context of general purpose computer systems.

Chapter 2 described the decision tree scheduling technique for handling conditional branch intensive scalar code. The DTS technique is a very general and robust code generation heuristic that efficiently utilizes concurrency in the form of parallelism and pipelining to reduce the average execution time of a tree of basic blocks. A key concept of the DTS technique is the use of guarded jump instructions to allow overlapped execution of multiple conditional branches, thus reducing the average delay of a conditional branch below that which can be provided by hardware. We have shown that the DTS technique, when combined with judicial code replication, achieves significant levels of speedup on a variety of example program modules.

For a sufficiently large decision tree and a sufficiently parallel machine, the DTS technique with guarded jumps and stores and selective code replication produces schedules that approach the theoretical speedup achievable on a highly parallel, no

overhead dataflow machine. Thus the DTS technique can be viewed as a static dataflow technique that captures many of the advantages of dataflow processing without incurring the inevitable overhead associated with dynamic dataflow processing.

Chapter 3 focuses on the problem of code generation for recurrence-intensive loop code. With the exception of job loads dominated by numerical computations, the use of linked data structures is pervasive in most general job loads. The traversal of linked data structures give rise to numerous recurrences through memory, reducing the effectiveness of vector and multiprocessor architectures. Horizontal architectures offer improved performance and cost-effectiveness; however horizontal architectures require sophisticated code generation techniques.

The the simple loop scheduling technique described in chapter 3 generates optimal throughput schedules for innermost loops without nested conditional statements. The SLS technique is an adaptation and extension of the theory of optimal design of hardware pipelines. We have shown that the SLS algorithm produces optimal throughput schedules in minimal time, i.e. the complexity of the SLS algorithm itself is optimal.

Architectural support for the proposed scheduling techniques is the subject of chapter 4. In this chapter we describe a highly concurrent parametric machine model that was used to develop the DTS and SLS techniques. We discuss the rationale behind the design decisions that lead to the choice of machine organization and architecture. We also discuss several related practical code generation problems including register assignment issues.

In conclusion, this thesis has

(i) pointed out some of the principle problems that must be solved in order to achieve high-speed general-purpose computing.

(ii) proposed new code optimization techniques for solving some of these problems, and

(iii) proposed a machine organization that supports these code optimization techniques and can be implemented using current technology.

5.2. Suggestions for Future Research

Although we have addressed some of the key problems of high-speed general-purpose computing, solutions to many more problems are necessary before a practical implementation of a computer system employing the techniques proposed in this thesis can be realized. Some of these problems are listed below.

Multilevel memory hierarchies are a standard feature of modern general-purpose computer systems. Throughout our research we have ignored the problem of page-fault handling. Since our techniques are targeted at large-scale application programs that require considerable computing and memory resources, a high-performance solution to the page-fault problem is essential in order to handle such applications.

Modern programming methodologies promote the use of many small procedures. We have ignored the problem of speeding up procedure calls in our research. The DTS technique can be easily extended to convert procedure calls to in-line expansion of the called procedure, at the cost of further increasing the amount of replicated code. However, it is much more desirable if in-line expansion can be limited only to those execution paths of a procedure that have a high probability of being taken in the context of the specific procedure call site. The use of intelligent procedure expansion techniques is expected to be crucial to the achievement of high performance for object-oriented programming methodologies that rely on extensive use of numerous small procedures.

As an extension of the guarded store and jump features, some jump instructions can actually be entirely eliminated by making subsequent guard expressions more complex. This possibility poses an interesting new optimization problem with a variety of tradeoffs, including code space and rescheduling opportunities. Further, rescheduling opportunities arise from attempting to use more detailed information about segment-by-segment pipeline operations that may relax dependency constraints on scheduling.

Finally, throughout this research we have concentrated exclusively on optimization techniques that exploit static information about the behavior of programs. It is well-known that much more precise information, and therefore superior optimization results, can be achieved if the dynamic behavior of a program is taken into account. We believe that static and dynamic optimization techniques are complementary, and the best solution in a system context should involve a combination of both types of techniques.

REFERENCES

[1] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, vol. 21, pp. 63-72, Jan. 1978.

[2] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, vol. 11, pp. 8-24, Jan. 1967.

[3] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, pp. 25-33, Jan. 1967.

[4] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Annual International Symposium on Computer Architecture*, pp. 135-148, 1981.

[5] G. Radin, "The 801 Minicomputer," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47, Mar. 1982.

[6] D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proc. 8th Annual Symposium on Computer Architecture*, pp. 443-458, May 1981.

[7] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Mar. 1982.

[8] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "MIPS: A Microprocessor Architecture," *Proc. 15th Annual Workshop on Microprogramming*, vol. 13, pp. 17-22, Oct. 1982.

[9] J. A. Fisher, "Very Long Instruction Word Architecture and the ELI-512," *Proc. 10th Annual International Symposium on Computer Architecture*, pp. 140-150, 1983.

[10] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, pp. 1405-1411, Dec. 1972.

[11] J. L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, vol. C-33, pp. 1221-1246, Dec. 1984.

[12] D. W. Clark and H. M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780," *Proc. 9th Annual Symposium on Computer Architecture*, pp. 9-17, Apr. 1982.

[13] T. R. Gross and J. L. Hennessy, "Optimizing Delayed Branches," *Proc. 15th Annual Workshop on Microprogramming*, vol. 13, pp. 114-120, Oct. 1982.

[14] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: Hemisphere Publishing Corporation, 1981.

[15] E. W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, pp. 453-457, Aug. 1975.

[16] D. J. Kuck, *The Structure of Computers and Computations*. New York: John Wiley and Sons, 1978.

[17] J. E. Thornton, "Parallel Operation in the Control Data 6600," *Proc. AFIPS Conference*, vol. 26, pp. 33-40, 1964.

[18] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976.

[19] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, vol. C-33, pp. 1023-1029, Nov. 1984.

[20] J. F. Thorlin, "Code Generation for PIE (Parallel Instruction Execution) Computers," *Proc. Spring Joint Computer Conference*, pp. 641-643, 1967.

[21] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478-490, Jul. 1981.

[22] E. W. Davis, Jr., "A Multiprocessor for Simulation Applications," Dept. of Computer Science Rep. UIUCDCS-R-72-527, University of Illinois at Urbana-Champaign, Urbana, IL, 1972.

[23] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *Proc. 9th Annual International Symposium on Computer Architecture*, pp. 113-119, Apr. 1982.

[24] *Cray-1 Reference Manual*. Minneapolis: Cray Research Inc., 1976.

[25] F. H. McMahon, *FORTRAN CPU Performance Analysis*. Livermore, CA: Lawrence Livermore Laboratories, 1972.

[26] S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Transactions on Computers*, vol. C-33, pp. 1013-1022, Nov. 1984.

[27] S.-C. Chen and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," *IEEE Transactions on Computers*, vol. C-24, Jul. 1975.

[28] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Transactions on Computers*, vol. C-31, pp. 363-376, May 1982.

[29] R. G. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," Dept. of Computer Science Rep. UIUCDCS-R-84-1177, University of Illinois at Urbana-Champaign, Urbana, IL, 1984.

[30] A. E. Charlesworth, "An Approach to Scientific Array Processing: the Architectural Design of the AP-120B/FPS-164 Family," *IEEE Computer*, vol. 14, pp. 18-27, Sep. 1981.

[31] B. R. Rau, C. D. Glaeser, and R. L. Picard, "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support," *Proc. 9th Annual International Symposium on Computer Architecture*, pp. 131-139, 1982.

[32] S. Even, *Graph Algorithms*. Maryland: Computer Science Press, 1979.

[33] J. H. Patel and E. S. Davidson, "Improving the Throughput of a Pipelines by Insertion of Delays," *Proc. 3rd Annual Symposium on Computer Architecture*, pp. 159-164, 1976.

[34] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, Mass.: Addison-Wesley, 1977.

[35] B. R. Rau, P. J. Kuekes, and C. D. Glaeser, "A Statically Scheduled VLSI Interconnect for Parallel Processors," in *VLSI Systems and Computations*. Computer Science Press, pp. 389-395, 1981.

[36] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.

VITA

Peter Yan-Tek Hsu was born on October 16, 1958 in Hong Kong. He left high school after the sophomore year to attend the University of Minnesota and received a Bachelor of Computer Science degree in 1979. He then entered the University of Illinois and received a Master of Science in Computer Science in 1982. Upon completion of the Doctor of Philosophy degree, he will join I.B.M. Research in Yorktown Heights, New York.

During his undergraduate studies at the University of Minnesota, Peter Hsu was a teaching assistant at the Department of Computer Science from 1976 to 1979 and also a research assistant at the Department of Psychology from 1977 to 1979. He was employed by Sperry Univac in the summer of 1979. While pursuing graduate studies at the University of Illinois, he was a teaching assistant at the Department of Computer Science from 1979 to 1980, and a research assistant at the Coordinated Science Laboratory from 1980 to 1985.