1-1-1991

The exploitation of pipeline parallelism by compile time dataflow analysis

Joseph Michael Lombardo
University of Nevada, Las Vegas

Order Number 1345926

The exploitation of pipeline parallelism by compile time dataflow analysis

Lombardo, Joseph Michael, M.S.

University of Nevada, Las Vegas, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

THE EXPLOITATION OF PIPELINE PARALLELISM BY COMPILE TIME DATAFLOW ANALYSIS

by

Joseph Michael Lombardo

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
April, 1991

The thesis of Joseph Lombardo for the degree of Master of Science in Computer Science is approved.

Evangelos Yfantis

Chairperson, Evangelos Yfantis Ph.D.

John T. Minor

Examining Committee Member, John Minor Ph.D.

Ajoy Kumar Datta

Examining Committee Member, Ajoy Datta Ph.D.

Graduate Faculty Representative, Rohan Dalpatadu Ph.D.

Graduate Dean, Ronald Smith Ph.D.

University of Nevada, Las Vegas
April, 1991

Abstract

The automatic and implicit transformation of sequential instruction streams, which execute efficiently for pipelined architectures is the subject of this paper. This paper proposes a method which maximizes the parallel performance of an *instruction pipeline* by detecting and eliminating specific *pipeline hazards* known as *resource conflicts*. The detection of *resource conflicts* is accomplished with data dependence analysis, while the elimination of *resource conflicts* is accomplished by instruction stream code *transformation*. The transformation of instruction streams is guided by *data dependence analysis*, and *dependence graphs*. This thesis is based on the premise that the elimination of *resource conflicts* is synonymous with the elimination of specific *arcs* in the dependence graph. Examples will be given showing how detection and elimination of *resource conflicts* is possible through compiler optimization.

Contents

List of Figures

Acknowledgments

I would like to thank my advisor and chairperson Evangelos Yfantis for his support and encouragement throughout my education and research at UNLV. Without his support this thesis would not have been possible. I would also like to to thank John Minor, Ajoy Datta and Rohan Dalpatadu for serving on my committee and for their helpful comments.

Chapter 1

Introduction

The goal of this thesis is to maximize the parallel performance of an *instruction pipeline* by utilizing compile time optimization. Pipelines are used to increase the *throughput* of a system by decreasing the *memory latency*. *Memory latency* is the elapsed time between the request for data by a processor and the receipt of that data by the processor. One cause of *memory latency* is the difference between memory cycle time and processor cycle time. Usually memory cycle time is greater than processor cycle time. *Throughput* is the number of tasks completely processed by the pipeline per unit of time [Dasgupta, 1989b].

This thesis proposes a series of compile time optimizations, based on dataflow analysis, that will detect and remove possible *resource conflicts* within an instruction stream. In its most general form, dataflow analysis is a method for finitely describing how a program utilizes its data [Muchnick and Jones, 1981]. For our purposes, dataflow analysis becomes a useful diagnostic tool that dis-

covers certain properties of a program before the program is executed. The information collected during dataflow analysis on an instruction stream can be utilized during the compiler optimization phase.

Chapter 2

Overview

2.1 Instruction Timing and Stalls

Instruction execution is accomplished through a series of steps called the **instruction cycle**. The *instruction cycle* consists of subcycles, each of which takes one or more clock cycles. Thus an instruction goes through a series of steps called cycles before actual execution. For our first examples an instruction cycle with 5 steps will be assumed with the following steps or cycles:

1. **IF**: The Instruction Fetch cycle fetches the next instruction from memory by loading the instruction register with the correct address of that instruction.

2. **ID**: The Instruction Decode cycle decodes the instruction and accesses the register file for a register fetch.

3. **EX**: The Execution cycle performs ALU operations or calculates an effective address.

4. **MEM**: The MEMory cycle is the only cycle which accesses memory.

5. **SR**: The Store Result cycle stores results back into the proper register.

In a **pipelined** system different instructions exist at different levels of the *instruction cycle* at the same time. That is, at time α, instruction I_i is executing while instruction I_j is fetching operands. Each **stage** of the pipeline completes a different part of the instruction cycle. An instruction enters at one end of the pipeline, proceeds through the different *stages* and exits at the other end. In a pipeline the *machine cycle* is the amount time needed to move an instruction one stage down the pipeline. The slowest cycle in the *instruction cycle* usually represents the length of the *machine cycle.*

	Machine Cycles								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	SR				
i+1		IF	ID	EX	MEM	SR			
i+2			IF	ID	EX	MEM	SR		
i+3				IF	ID	EX	MEM	SR	
i+4					IF	ID	EX	MEM	SR

When a pipeline is free of hazards an instruction is executed every *machine cycle* as shown in figure 2.1. **Pipeline hazards** are the conditions

Figure 2.1: An example of a pipeline.

within a pipelined system that disrupt, delay, or prevent the smooth flow of tasks through the pipeline. One type of *pipeline hazard* is referred to as a **resource conflict**. A *resource conflict* is created when two instructions try to access the same *resource* at the same time. Normally when a *resource conflict* is detected, the pipeline will **stall** allowing one instruction to complete before the next instruction is allowed access to the resource. A *stall* may take one or more machine cycles to complete thus reducing the efficiency of a pipeline.

An example of such a hazard can be seen in figure 2.2. The pipeline shown in figure 2.2 shares a single data-memory reference port. That is, when an instruction is using that data-memory port it has complete access control over the port until it is finished with the port. In our instruction cycle the MEM and IF steps both use the data-memory port when fetching data from memory. In figure 2.2 at machine cycle time 4 a stall occurs because the *load* instruction needs an extra cycle to complete the data transfer during the *MEM* stage before instruction i+3 can be fetched.

	Machine Cycles								
	1	2	3	4	5	6	7	8	9
LOAD	IF	ID	EX	MEM		SR			
i+1		IF	ID	EX	MEM	SR			
i+2			IF	ID	EX	MEM	SR		
i+3				*stall*	IF	ID	EX	MEM	SR
i+4						IF	ID	EX	MEM

Figure 2.2: A pipeline hazard and stall at a cost of one machine cycle.

Some instruction delays can cost as much as 19 cycles and are usually found in the floating-point instruction set. In figure 2.3 a 19-cycle delay for two consecutive floating point divide instructions is shown. In figure 2.3 there is only one floating point functional unit and the second instruction must wait 19-cycles before it can access the floating-point unit.

	Machine Cycles								
	1	2	3	...	22	23	...	41	42
FDIV	IF	ID	EX		MEM	SR			
FDIV		IF	ID		EX			MEM	SR

Figure 2.3: A pipeline hazard and stall at a cost of 19 machine cycles.

Each of the following subsections describe a particular pipelined machine and the instruction delays found in that machine.

2.1.1 The MIPS R2000

The MIPS R2000 has a 5-stage pipeline, with 5 active instructions in the pipeline at any time. One instruction is started down the pipeline every machine cycle. The MIPS R2000 instructions are divided into four groups:

1. Computational : All register to register with two or three operands.

2. Load or Store: The only instructions that are allowed access to memory. Load and Store instructions have a **1 cycle delay** before data being transferred is available to another instruction.

3. Jump and Branch: Relative jumps, straight jumps and compares. Jump and Branch instructions have a **1 cycle delay** while they fetch the instruction and the target address.

4. Special Instructions; These instructions support procedure and interrupt linkage.

The MIPS R2010 Floating-Point Accelerator (FPA) operates as the coprocessor for the R2000 and has a 6-stage pipeline. The MIPS R2010 instructions are divided into four groups:

1. Computational : All register to register with two or three operands.

- **ADD and SUB**: 2-cycle delay.
- **MUL.S**: 4-cycle delay.
- **MUL.D**: 5-cycle delay.
- **DIV.S**: 12-cycle delay.
- **DIV.D**: 19-cycle delay.

2. Load, Store and Move: The only instructions that are allowed access to memory. Load and Store instructions have a **2-cycle delay** before data being transferred is available to another instruction.

3. Conversion: Performs conversion operations between the various data formats. Delays of 5-cycles are possible.

4. Compare ; Performs comparisons between registers and sets condition bits. Delays of 5-cycles are possible.

2.1.2 The Stanford MIPS

The Standford MIPS has a 5-stage pipeline, with three active instructions in the pipeline at any time. One instruction is started down the pipeline every two clock cycles. All instructions execute in one machine cycle. The MIPS instructions are divided into four groups:

1. ALU : All register to register with two or three operands. A total of 13 instructions in this group.

2. Load or Store: The only instructions that are allowed access to memory. A total of 10 instructions in this group.

3. Control Flow: Relative jumps, straight jumps and compares. A total of 6 instructions in this group.

4. Special Instructions; These instructions support procedure and interrupt linkage. A total of 2 instructions are found in this group.

2.1.3 The IBM 801

Memory on the IBM 801 is accessed by the *Load and Store* instructions. Multiplication is supported by a MULTIPLY STEP instruction which uses 16 clock cycles and division is supported by a DIVIDE STEP which uses 32-cycles. All other instructions execute in one machine cycle.

2.1.4 The GMU Microcoded RISC-MIRIS

The MIRIS has a set of 64 primitive instructions and each instruction executes in a single machine cycle.

2.1.5 The ACORN RISC Machine

There are 44 basic instruction codes which are subdivided into 5 main groups. All instructions except the multiple register load and store execute in one cycle. The groups are:

1. Load or Store: Single register.

2. Load or Store: Multiple registers

3. ALU: all register to register.

4. Branch

5. Software interrupt

2.2 Data Reference and Stalls

Another version of a *resource* conflict is the data reference conflict. A data reference conflict occurs when the order in which operands are accessed is changed by the pipeline. A data reference can be seen in figure 2.4. Instruction i must store the new value of R1 before instruction i+1 is allowed a register fetch of R1. If the order is changed instruction i+1 will have the old value of R1 not the value that was placed into it by instruction i.

i	MOV R4,R1	(R1 := R4)
$i+1$	ADD R1,R2,R2	(R2 := R1 + R2)

	Machine Cycles								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	SR				
i+1		IF				ID	EX	MEM	SR

Figure 2.4: Resource conflict on R1

2.3 Branching and Stalls

A natural characteristic of a pipeline is the ability to prefetch one instruction, while a previous instruction is being executed. When the executed instruction is a successful branch or an unconditional one, the prefetched instructions must be *flushed* from the pipeline. When *flushing* occurs in a pipeline the *flushed* instructions add to the wasted memory access time, and the computing time for a task increases.

Figure 2.5 is an example of a branching stall in a pipeline which has the ability to hold four instructions.

$i+20$	JMP 106
$i+21$	ADD R4,R5,R3
$i+22$	ADD R4,R6,R2
$i+23$	STORE R3,[R5+R7]

	Machine Cycles								
	40	41	42	43	44	45	46	47	
i+20	IF	ID	EX	MEM	SR				
i+21		IF	ID	EX	MEM	SR			
i+22			IF	ID	EX	MEM	SR		
i+23				IF	ID	EX	MEM	SR	
i+24					IF	ID	EX	MEM	SR

	Machine Cycles									
	40	41	42	43	44 ***	50	51	52	53	54
i+20	IF	ID	EX	MEM	SR					
i+106						IF	ID	EX	MEM	SR
i+107							IF	ID	EX	MEM
i+108								IF	ID	EX
i+109									IF	ID

Figure 2.5: A pipeline stall due to branching.

Notice in figure 2.5 that the pipeline was filled with five instructions

where the the first instruction, i+20, is an unconditional branch to instruction i+106. The next three instructions, i+21,i+22 and i+23 must be flushed from the pipeline and instruction i+106 is fetched at a cost of 6 machine cycles. Although this may not seem like a large problem we know that 65% of control instructions change the value of the PC [Hennessy and Patterson, 1990].

2.4 Previous Work

The following is an overview of pipeline scheduling solutions. Most of the solutions are based on the the concept of a Directed Acyclic Graph (DAG).

- **NP–Complete** : In 1983, J.Hennesey and T. Gross prove that code reorganization for an optimal pipeline is NP-Complete [J.Hennesey and Gross, 1983]. Along with the proof, an algorithm is given which purposes a solution to the code reorganization problem. The algorithm works on a DAG which incorporates a *look–ahead* scheme for node scheduling. A major problem with this concept is the algorithm itself can deadlock during scheduling.

- **Instruction Scheduling for Vector Processors**: Described in his paper [Aray, 1985], S. Aray's algorithm uses a weighted DAG to solve the code scheduling problem of a vector processor. Since the time complexity of this solution is exponential, the use of this algorithm in a compiler is not feasible.

- **Gibbons Method** : A reorganizational scheme known as the Gibbons Method, is described in a paper by P.B. Gibbons and S. Muchnick [Gibbons and Muchnick, 1986]. This method is based on a DAG representation of instructions and a *Candidate Set.* The *Candidate Set* represents instructions that are ready for scheduling, that is the instructions with no predecessors in the DAG.

- **Critical Path** : An algorithm that works on a unweighted DAG, where nodes represent instructions and the arcs represent dependencies is presented by D. Berstien in 1988 [Berstien, 1988]. This algorithm follows the *critical path approach* where each instruction is assigned to a level in the DAG. The nodes in the graph are then arranged in descending order where instructions with the highest level are scheduled first. In 1989 Bernstien [D.Bernstien and Gertner, 1989a] [D.Bernstien and Gertner, 1989b] extends the algorithm so that it works on a weighted DAG, where the weights represent delay slots of zero or one. Thus the largest delay possible with this algorithm is one cycle.

- **The GNU Instruction Scheduler** : The GNU compiler incorporates the *critical path* concept where each instruction is given a priority [Tiemann, 1989] based on path length and the execution time of the instruction. An instruction that takes longer to execute will be given a higher priority, and the instruction with the highest priority is scheduled first.

- **The MIPS Reorganizer** : The MIP–X compiler uses a three stage DAG reorganizing process : (1) local reorganization,(2) interblock reorganization , and (3) branch scheduling [Chow, 1989]. The local reorganizer detects the branch instructions, thus defining the basic blocks. Where as the interblock reorganizer determines when an instruction can be moved up beyond a basic block boundary to fill delay slots avoiding a possible NOP instruction fill.

2.5 Survey of Dataflow Analysis

- **Reaching Definitions**: Dataflow analysis was first used by Vyssotsky in 1961 as a compile time diagnostic tool for the Bell Laboratories IBM 7090 FORTRAN II compiler [Hecht, 1977]. Vyssotsky used dataflow analysis to solve the *reaching definitions* problem. If there exist two blocks, B_i and B_j, then a definition d, defined in B_i, is said to *reach* B_j if d is not redefined between B_i and B_j.

- **Variable Folding** : In 1969 E.S. Lowry and C.W. Medlock [Lowry and Medlock, 1969] implemented a *variable folding* dataflow analysis algorithm. When an instruction has the form of $X := Y$, we substitute the value of Y for any future undefined *uses* of X.

- **Instruction Scheduling** : In 1970 [Sethi et al., 1970] and 1974 [Beatty, 1972] dataflow analysis was utilized for the optimization of arithmetic expres-

sions. Various target architectures often follow different scheduling criteria. *Instruction scheduling* is a method of mapping the program code to a specific architecture. This optimization often leads to performance gains. [Muchnick and Jones, 1981]

- **Register Allocation** : J.C. Beatty [Beatty, 1974] in 1974 proposed a dataflow analysis algorithm that attempts to eliminate useless temporary variables by assigning program variables to CPU registers.

- **Dead Code Elimination** : The instruction $X := Y$, given as an example in the *variable folding* section, would be detected as *dead code* and eliminated after constructing *use-definition chains* during dataflow analysis. *Use-definition chains* or *ud-chains* are lists of each *use* of a variable and all the *definitions* that reach that variable. This technique was first used by K.Kennedy at Rice University in 1975 [K.Kennedy, 1975].

- **Detection of Parallelism** : In 1975 P.B. Schneck [Schneck, 1975], with the use of dataflow analysis, detected and coded implicit parallel vector expressions.

- **Formal Data Dependence**: By 1976 U.Banerjee [Banerjee, 1976] [Banerjee, 1979] had discovered three of the most popular dependence tests: gcd, bound, and inequality. Banerjee's work has become a foundation in data dependence analysis [Wolfe, 1982] which has been modi-

fied for many purposes by several authors [Kennedy, 1984] [Allen and Kennedy, 1982] [Ellis, 1985] [Wolfe, 1982].

- **Recursive Data Structure Analysis**: In 1977 ,Jones and Muchnick proposed a general framework for dataflow analysis on programs with recursive data structures [N.D.Jones and S.Muchnick, 1982].

- **Dependence Direction**: In his 1982 Ph.D. thesis M. Wolfe [Wolfe, 1982] discovered the *direction vector*. The discovery of the *direction vector* lead Wolfe to several important tests which recognize parallelism within a loop. .

    1. The *vectorization test* recognizes whether the statements in a loop can be vectorized. After building the DDG [1] the compiler attempts to find cycles and backward *direction vectors* within the loop. The loop is vectorizable if it is void of any cycles and backward *direction vectors*. If the loop is represented by a backward *direction vector* the elimination of all upward arcs by code reordering is attempted. The topological sorting sometimes reverses the *direction vector*, from backwards to forwards, allowing vectorization of the loop.

    2. The *loop fusion test* recognizes a situation where two or more loops in a program can be transformed into a single vectorized loop.

[1] An as example is given in 2.6.

3. The *loop interchanging test* recognizes the possibility of interchanging nested loop levels. In certain architectures, an increase in performance can be seen when using this technique [Kennedy, 1984].

- **Subscripted Variable Analysis**: By modifying Banerjee's work in 1983 J. Allen [Allen, 1983] showed how data dependence analysis could exploit parallelism at the loop level by the dataflow analysis of array subscripts.

- **Loop Interchange**: An early example of a loop transformation, based on Banerjee's data dependence analysis technique, is referred to as *loop interchange* and was discovered by J. Allen and K. Kennedy [Kennedy, 1984] in 1984.

- **Minimization of Communication** : C.D. Polychronopoulos utilized dependence analysis in 1987 for the reduction of interprocess communication in a message passing system [Polychronopoulos, 1987a]. Through analysis of the data dependence graph [2], transformations are selected which reduce the total number of messages needed in a parallel program.

- **Pointer Analysis**: In 1989, S. Horwitz, P. Pfeiffer, and T. Reps, discovered a method for analyzing data dependence for pointer variables.

[2]See section 2.6

2.6 Graphs

2.6.1 Data Dependence Graphs, DDG

As the compiler computes data dependence information, it create a data dependence graph, or DDG. A DDG is a directed graph $G = (V, E)$, where the nodes, $V = S_1, S_2, ..., S_n$, represent the statements in a program, and the directed arcs, $E = \{e_{ij} = (S_i, S_j) \mid S_i, S_j \in V\}$, represent the dependence relationships. Parallelism is extracted after the creation of the DDG and the code transformation phase begins. The four classifications of dependence are shown in figure 2.6

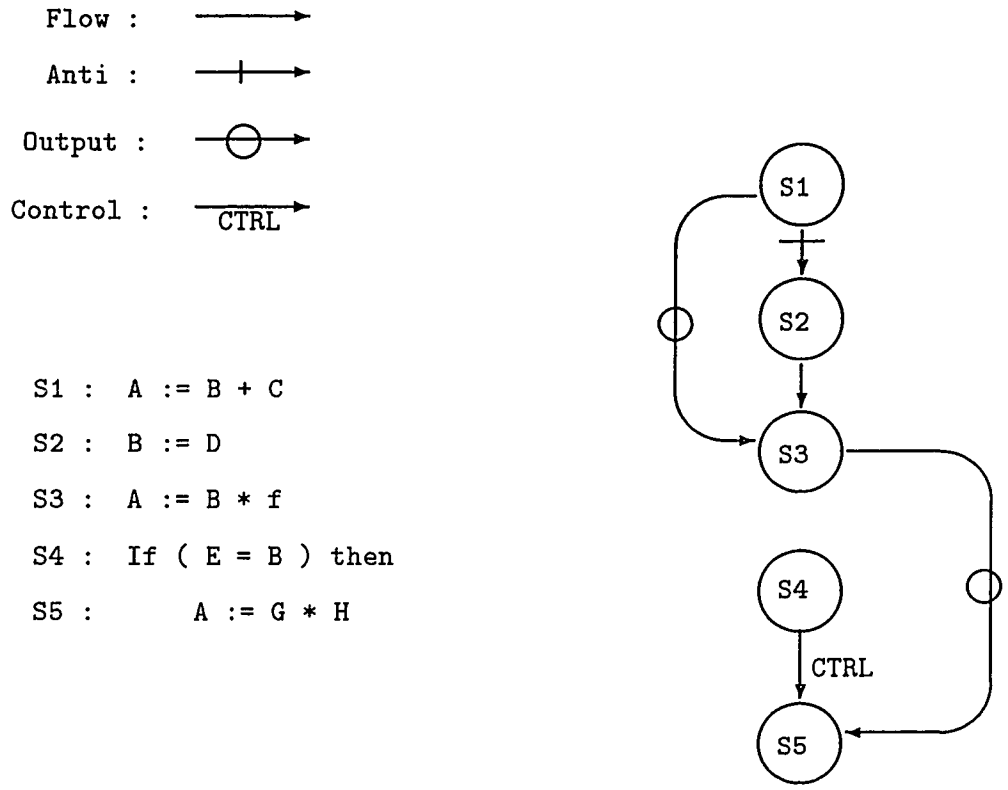


Figure 2.6: Data Dependent Graph, DDG

2.6.2 Iteration Space Dependence, ISDG

The analysis of data dependence for the recognition of parallelism at a statement or block level is very useful. Although dependence tests within loops do not necessarily find true dependence [Polychronopoulos, 1987b], we can

compute **diophantine equations** to find any dependence within loop iterations. The analysis at a subscript level will determine data dependence within a loop construct. Different iterations of a loop can be run in parallel if, and only if, the loop carries no dependence between the iterations. This type of dependence is referred to as **loop-carried** .

In loop constructs the compiler writer needs to form and solve **dependence equations** that involve array subscripts. Solving this dependence is equivalent to solving a **diophantine equation**. Various methods used in solving these equations are given in [Banerjee, 1979] [Wolfe, 1982] [Allen and Kennedy, 1982] [Griffin, 1954] [Kirch, 1974]. The only values valid for our analysis are integer values in the range of the loop counter.

The **dependence distance** gives the number of loop iterations between the corresponding dependent array elements. Distance is represented by an integer value. With distance and direction computed a ISGD, iteration space graph, is created.

The **dependence direction** shows the relationship between instances of each loop iteration. Direction is represented by the >, <, and = operators. The following is a summary:

- = : Dependence holds within the same loop iteration.
- > : Dependence holds from a particular loop iteration back to a previous loop iteration when the computed value of the distance vector is larger than 0. This is denoted in the dependence graph with a down-

ward arc.

- $<$: Dependence holds from a particular loop iteration to a future loop iteration when the computed value of the distance vector is less than 0. This is denoted in the dependence graph with a upward arc.

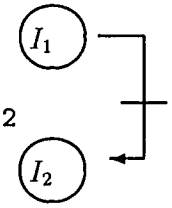
- $*$: Depedence is unknown or all three, $<$, $>$, $=$, apply.

Consider the next few examples:

Example 1

$I_1 \overline{\delta}_{(>)} I_2$

```
   DO i = 1 to N      equation:  i=j-1
I1:    A(i)=          possible solution.  i=1 j=2
I2:       =A(i-1)
   ENDDO              distance:  j-i=1
                      direction:  j-i>0
```



Example 2

$I_1 \overline{\delta}_{()} I_2$

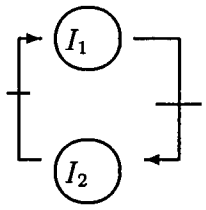
```
   DO i = 1 to N equation:  2i=2j-1
I1:    A(2i)=          No solutions no
I2:       =A(2i-1)        dependence
   ENDDO
```

Examples 1 and 2 deal with single dimensional arrays. The same techniques are used in multidimensional arrays. The difference is that each dimension is solved separately. In the case of two dimensions there are two **dependence equations**, two **distances**, and two **directions**.

Example 3 Multidimensional subscripts

$I_1 \overline{\delta}_{(>,<)} I_2$

```
   DO 10 i=1 to N
     DO 20 j=1 to M
I1:     A(i,j)=
I2:         =A(i-1,j+1)
     20
   10
```




```
equation1 :  i = k-1
distance 1:  1       direction 1:  >
equation 2:  j=1+1
distance 2:  --1     direction 2:  <
```

Usually, in multiple subscripts, **distance** and **direction** are combined into a vector form . In example 3 the **distance vector** is represented by (1, -1), and the **direction vector** is represented by (>, <).

Informally, a loop can be described by an iteration space.[Wolfe, 1982] Thus, a *d*–dimensional loop is described by a *d*–dimensional iteration space.

2.7 Notation and Definitions

- $\mathbf{IB_n}$: ... a *basic block* of n instructions $\mathbf{IB_n} \equiv \{\mathbf{I_1}, \mathbf{I_2}, ..., \mathbf{I_n}\}$.
- $\mathbf{I_i} < \mathbf{I_j}$: ... when instruction $\mathbf{I_i}$ lexically precedes instruction $\mathbf{I_j}$ in the instruction stream.
- **Instruction Formats** ... memory is access only by the LOAD and

STORE operations.

ADD scr1,scr2,dst	$\equiv$	dst := scr1 + scr2
SUB scr1,scr2,dst	$\equiv$	dst := scr1 - scr2
LOAD [src1+scr2],dst	$\equiv$	dst := memory[src1+scr2]
MOV src1,dst	$\equiv$	dst := src1
STORE scr1,dst	$\equiv$	memory[dst] := scr1
JMP dst	$\equiv$	PC := dst
INC src1	$\equiv$	src1 := src1 + 1

- $\mathbf{I_i \delta^* I_j}$ denotes **any dependence.**
- $\mathbf{I_v \Delta I_w}$ denotes **indirect dependence** , if
  $\mathbf{I_v \delta^* I_{v_1} \delta^* ... \delta^* I_{v_n} \delta^* I_w}$ then $\mathbf{I_v \Delta I_w}$
- $\mathbf{I_i \Delta I_i}$ denotes a **cyclic dependence** usually found in loops.
- **IN($\mathbf{I_i}$)** ...the set of variables read by instruction $\mathbf{I_i}$. Consider the following instruction :
  $\mathbf{I_i}$ MOV R1,R2
  IN($\mathbf{I_i}$) = { R1 }
- **OUT($\mathbf{I_i}$)** ... the set of variables written to by instruction $\mathbf{I_i}$. Consider the following instruction :

$\mathbf{I_i}$ MOV R1,R2

$\mathbf{OUT(I_i)}$ = { R2 }

- $\mathbf{P_{s,a}}$ ... a pipeline where s = the number of stages and a = the number of active instructions allowed in the pipeline at anyone time. Thus $P_{5,3}$ denotes a pipeline with 5–stages which has the ability to hold 3–active instructions at anyone time.

- **Pipeline Formats** ... We will use three different types of pipeline structures in our examples. The first type is a 5–stage pipeline, which has the ability to hold five active instructions. The stages are listed below:

  1. **IF**: The Instruction Fetch cycle fetches the next instruction from memory by loading the instruction register with the correct address of that instruction.
  2. **ID**: The Instruction Decode cycle decodes the instruction and accesses the register file for a register fetch.
  3. **EX**: The Execution cycle performs ALU operations or calculates an effective address.
  4. **MEM**: The MEMory cycle is the only cycle which accesses memory.
  5. **SR**: The Store Result cycle stores results back into the proper register.

The second pipeline type is a 5–stage pipeline, which has the ability to hold five active instructions. The stages are listed below:

1. **IF**: The Instruction Fetch cycle fetches the next instruction from memory by loading the instruction register with the correct address of that instruction.
2. **ID**: The Instruction Decode cycle decodes the instruction.
3. **OD**: The Operand Decode cycle calculates an effective address and fetches operands.
4. **SX**: The Store/Execute cycle sends operand to memory or uses ALU if execution.
5. **OF**: Operand Fetch if the instruction is a load.

The third pipeline type is a n–stage pipeline, which has the ability to hold a–active instructions. The stages are listed below:

- S_1:
- S_2:

  $\vdots$
- S_n:

Chapter 3

Instruction Pipeline Optimization via Dataflow Analysis

This chapter will discuss a technique used in optimizing instruction pipelines with the use of dataflow analysis. The technique analyzes the structure of a *basic block* of instructions [1] and detects data dependence that might create a *resource conflict* within the pipeline.

When two operands reference the same location in memory a dependence relation must be recognized [Allen, 1986]. To determine whether two operations have the ability to execute in parallel requires data dependence analysis [Padua and Wolfe, 1986]. Data dependence analysis at the statement or loop level reveals *fine grain* parallelism. Analysis at the subprogram or block level reveals *coarse grain* parallelism. Architectures tend to perform bet-

[1] A *basic block* of instructions is a straight line sequence of instructions within which the existence of a branch instruction may appear only as the last instruction in the block [Sethi and Ullman, 1986].

ter in either *fine grain* or *coarse grain* environments. Compiler writers, guided by specific architectures, choose the type of granularity they need. The different classifications of data dependence are *flow, anti, output,* and *control.*

3.1 Pipeline Dependence

3.1.1 Flow Pipeline Dependence

Definition 3.1.1 $\mathbf{I_i \delta I_j}$ *:* $\mathbf{I_j}$ *is* **flow dependent** *on* $\mathbf{I_i}$. $\mathbf{I_i}$ *must store its results before* $\mathbf{I_j}$ *is allowed to execute. Flow dependence exists in pipeline* $P_{s,a}$ *iff*

$(|\mathbf{i} - \mathbf{j}| < \mathbf{a}) \wedge (\mathbf{I_i} < \mathbf{I_j}) \wedge (\mathbf{OUT(I_i)} \bigcap \mathbf{IN(I_j)} \neq \emptyset)$

i	MOV R4,R1	(R1 := R4)
$i+1$	ADD R1,R2,R2	(R2 := R1 + R2)

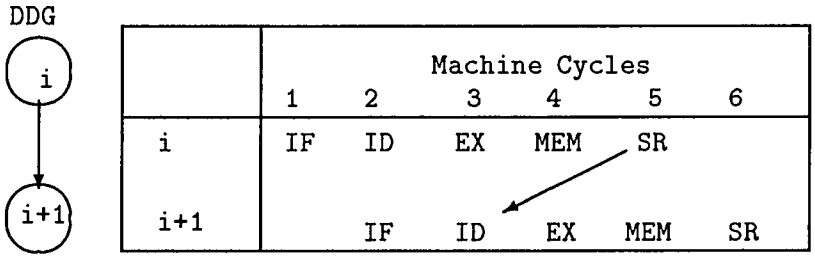


Figure 3.1: Flow Pipeline Dependence

A *flow dependence* can be seen in figure 3.1. At time 3, instruction $\mathbf{i+1}$ will access R1 for a read, while instruction $\mathbf{i}$ will not write that value until time 5 . Notice the formal condition holds since

$$(|\mathbf{i-j}| < \mathbf{a}) \wedge (\mathbf{i} < \mathbf{i+1}) \wedge (\mathbf{OUT(i)} = \{\mathbf{R1}\} \bigcap \mathbf{IN(i+1)}\{\mathbf{R1}\} = \{\mathbf{R1}\} \neq \emptyset)$$

3.1.2 Anti Pipeline Dependence

Definition 3.1.1 $\mathbf{I_i \bar{\delta} I_j}$ *:* $\mathbf{I_j}$ *is* **anti dependent** *on* $\mathbf{I_i}$. $\mathbf{I_j}$ *must fetch its data before* $\mathbf{I_i}$ *is allowed to change that value. Anti dependence exists in pipeline* $P_{s,a}$ *iff*

$$(|\mathbf{i-j}| < \mathbf{a}) \wedge (\mathbf{I_i} < \mathbf{I_j}) \wedge (\mathbf{IN(I_i)} \bigcap \mathbf{OUT(I_j)}) \neq \emptyset$$

i	LOAD mem[R1],R4	(R4 := mem[R1])
$i+1$	STORE R6,mem[R1]	(mem[R1] := R6)

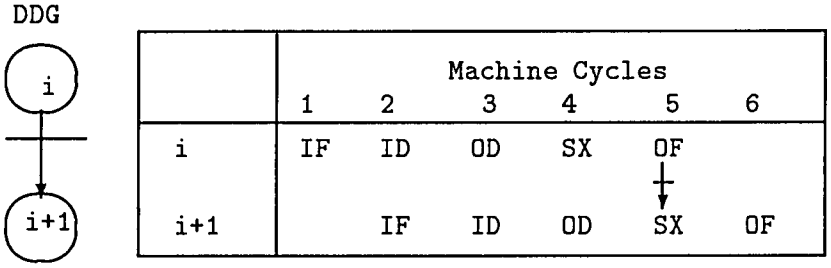


Figure 3.2: Anti Pipeline Dependence

An anti pipeline dependence can be occur in a pipeline that has the ability to *write* in an earlier stage than a *read.* In figure 3.2 instruction **i** attempts to read mem[R1], while instruction $\mathbf{i+1}$ attempts a write of mem[R1]. If the original order is not preserved instruction i will have the wrong value in R4. Notice the formal condition holds since

$$(|\mathbf{i}-\mathbf{j}| < \mathbf{a}) \wedge (\mathbf{i} < \mathbf{i}) \wedge (\mathbf{IN(i)} = \{\mathbf{R1}\} \bigcap \mathbf{OUT(i+1)}\{\mathbf{R1}\} = \{\mathbf{R1}\} \neq \emptyset)$$

3.1.3 Output Pipeline Dependence

Definition 3.1.1 $\mathbf{I_i}\delta^{\mathbf{O}}\mathbf{I_j}$ *:* $\mathbf{I_j}$ *is* **output dependent** *on* $\mathbf{I_i}$. $\mathbf{I_i}$ *must store its results before* $\mathbf{I_j}$ *is allowed to store its results.*

Output dependence exists in pipeline $P_{s,a}$ *iff*

$$(|\mathbf{i}-\mathbf{j}| < \mathbf{a}) \wedge (\mathbf{I_i} < \mathbf{I_j}) \wedge (\mathbf{OUT(I_i)} \bigcap \mathbf{OUT(I_j)} \neq \emptyset)$$

i	MOV R4,R1	(R1 := R4)
$i+1$	INC R1	(R1 := R1 + 1)

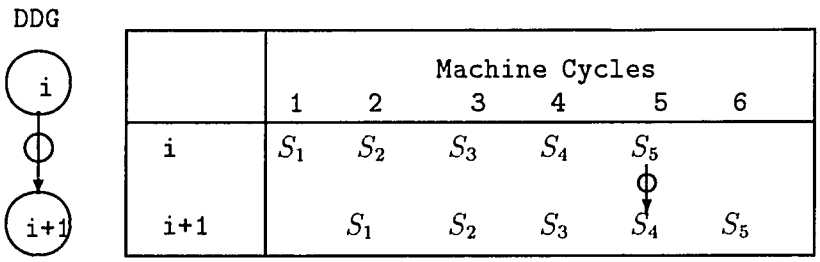


Figure 3.3

Figure 3.3: Output Pipeline Dependence

An output pipeline dependence can be occur in a pipeline that has the ability to *write* in two or more stages . In figure 3.3 instruction **i** attempts a write of *variable*, while instruction **i + 1** attempts a write of *variable*. If the original order is not preserved instruction i will have the wrong value for *variable*. Notice the formal condition holds since

$$(|\mathbf{i}-\mathbf{j}| < \mathbf{a}) \wedge (\mathbf{i} < \mathbf{i+1}) \wedge (\mathbf{OUT(i)} = \{\mathbf{var}\} \bigcap \mathbf{OUT(i+1)}\{\mathbf{var}\} = \{\mathbf{var}\} \neq \emptyset)$$

3.2 Pipeline Iteration Space Dependence

The exploitation of parallelism is strongly influenced by the characteristics of a language paradigm. A rich source of parallelism in the imperative languages can be found in the loop construct. Imperative languages are notorious for hiding data dependencies because of variable aliasing, side effects, and loop constructs [Ackerman, 1981] [D'Hollander and Opsommer, 1987].

Our method will first concentrate on breaking data dependence within looping structures. We recognize the fact that the majority of computing time is spent in loop computation. With our scheme, some types of architectural bottlenecks created by loops can be detected and eliminated from the instruction stream. *Data dependence direction vectors and distance vectors* are used to compute dependence within a loop structured iteration space. We look at both single and multiple subscripts.

The following sections will illustrate how dependence information can aid in the transformation of sequential loops into parallel code.

3.2.1 Dependence Free Loops

For our purposes there are two cases in which we will not need to make any loop transformations. The first occurs when the **distance vectors** , and the **direction vectors** are computed and the outcome is such that dependence does not exist. A loop void of data dependence between its iterations needs no further transformations.

The second occurs when the **distance vectors** , and the **direction vectors** are computed and the **distance vector** is larger than the number of active instructions in the pipeline. Any two statements that are related by this dependence can never be in the pipeline at the same time.

3.2.2 Loop Interchange

Loop interchange is the process of interchanging the nested depth of a nested loop. In certain architectures, an increase in performance can be seen when using this technique [Kennedy, 1984]. Loop interchanging reorders the original sequence of statements. As discussed previously, we must know the dependence information before we can perform any transformations. If the dependence directions are $(<,>)$, this technique is impossible [Wolfe, 1982].

Requirements for loop interchanging :

1. The loops must be tightly nested.

2. The loop limits of inner and outer loops must be invariant.

3. There is no dependence relation $\mathbf{S_i}\delta^*_{(<,>)}\mathbf{S_j}$

Example 3.2 shows a nested loop before and after loop interchange.

Example 3.2

Before Transformation

DO 10 I= 2 to N

DO 20 J = 2 to M

A(I,J) = A(I, J - 1)

20

10

Dependence Equation 1: $(=,>)$

```
A(2,2) = A(2,1)
   \
A(2,3) = A(2,2)        Flow Dependence
   \
A(2,4) =A(2,3)
   \
    etc.
```

After Transformation

DO 10 J= 2 to M

DO 20 I = 2 to N

A(I,J) = A(I, J - 1)

20

10

Dependence Equation 1: (>,=)

```
A(2,2)=A(2,1)

A(3,2)=A(3,1)        dependence is broken

A(4,2)=A(4,1)

  etc.
```

3.2.3 Loop Code Motion

Loop statement reordering is the process of interchanging certain statements within a single loop. This technique reorders the original sequence of statements. As discussed previously, we must know the dependence information before we can perform any transformations. If the dependence direction is (=) the technique is impossible [Wolfe, 1982].

Requirements for loop code motion :

1. The loops must be a single loop or the most inner nested loop.

2. There is no dependence relation $\mathbf{S_i}\delta^{*}_{(=)}\mathbf{S_j}$

Example 3.3 shows a loop before and after code motion

Example 3.3

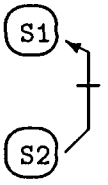
Before Transformation $S_1\overline{\delta}_{(<)}S_2$

```
 DO 10 I = 1 to N
   A(I)=C(I)
   D(I)=A(I+1)
 10
```

```
A(1)=C(1)
D(1)=A(2)

A(2)=C(2)
D(2)=A(3)

A(3)=C(3)
D(3)=A(4)
  etc.
```



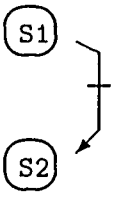
After Transformation $S_1\overline{\delta}_{(<)}S_2$

```
 DO 10 I = 1 to N
   D(I)=A(I+1)
   A(I)=C(I)
10
```

```
D(1)=A(2)
A(1)=C(1)

D(2)=A(3)
A(2)=C(2)

D(3)=A(4)
A(3)=C(3)
```



3.3 Pipeline Dependence Graph , PDG

Since pipelines are machine dependent, a certain amount of information must be known by the compiler for an effective analysis phase. This section will show how the PDG is constructed.Parallelism is extracted after the creation of the PDG by the code transformation phase.

As the compiler computes data dependence information, it create a pipeline dependence graph, or PDG. A PDG is a directed graph $G = (V, E)$, where the nodes, $V = S_1, S_2, ..., S_n$, represent the statements in a program, and the directed arcs, $E = \{e_{ij} = (S_i, S_j) \mid S_i, S_j \in V\}$, represent the dependence relationships,iteration space dependence,delay branching dependence and instructional timing dependence.

3.3.1 Data Dependence Graph

The approach begins when the compiler computes the IN() OUT() sets for each statement in the basic block[Polychronopoulos, 1987a]. Where $\mathbf{IB_n}$ is a *basic block* of n instructions, $\mathbf{IB_n} \equiv \{\mathbf{I_1}, \mathbf{I_2}, ..., \mathbf{I_n}\}$, and $\mathbf{I_i} < \mathbf{I_j}$ when instruction $\mathbf{I_i}$ lexically precedes instruction $\mathbf{I_j}$ in the instruction stream. The compiler extracts the dependence information by using the following definitions;

Flow Dependence

$$(\mathbf{I_i} < \mathbf{I_j}) \wedge (\mathbf{OUT(I_i)} \bigcap \mathbf{IN(I_j)}) \neq \emptyset$$

Anti-Flow Dependence

$$(\mathbf{I_i} < \mathbf{I_j}) \wedge (\mathbf{IN(I_i)} \bigcap \mathbf{OUT(I_j)}) \neq \emptyset$$

Ouptut Dependence

$$(\mathbf{I_i} < \mathbf{I_j}) \wedge (\mathbf{OUT(I_i)} \bigcap \mathbf{OUT(I_j)}) \neq \emptyset$$

The segment of code found in figure 3.4 will help us illustrate scalar data dependence.

$\mathbf{I_1}$	MOV R3,R1	$IN(1) = \{R3\}$	$OUT(1) = \{R1\}$
$\mathbf{I_2}$	ADD R1,R2,R2	$IN(2) = \{R1, R2\}$	$OUT(2) = \{R2\}$
$\mathbf{I_3}$	FDIV R9,R2,R4	$IN(3) = \{R9, R2\}$	$OUT(3) = \{R4\}$
$\mathbf{I_4}$	FDIV R9,R8,R6	$IN(4) = \{R9, R8\}$	$OUT(4) = \{R6\}$
$\mathbf{I_5}$	ADD R9,R2,R0	$IN(5) = \{R9, R2\}$	$OUT(5) = \{R0\}$
$\mathbf{I_6}$	LOAD [R9+R0],R7	$IN(6) = \{R9, R0, mem[R9 + R0]\}$	$OUT(6) = \{R7\}$
$\mathbf{I_7}$	STORE R7,[R11]	$IN(7) = \{R7, R11\}$	$OUT(7) = \{mem[R11]\}$
$\mathbf{I_8}$	JMP 137		

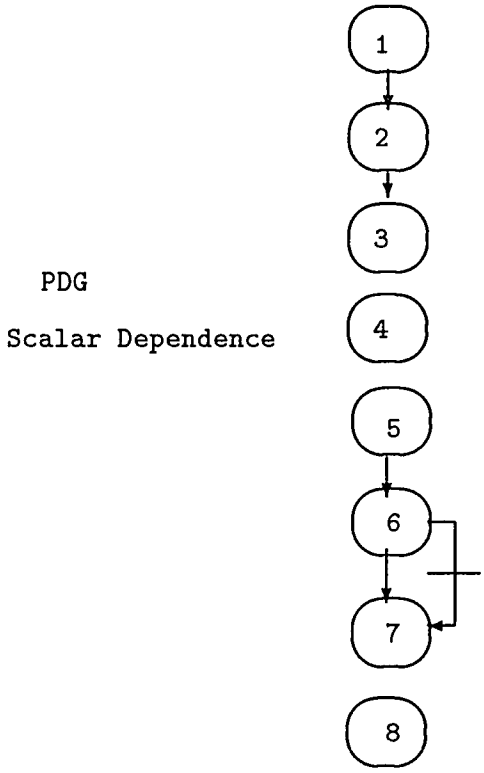


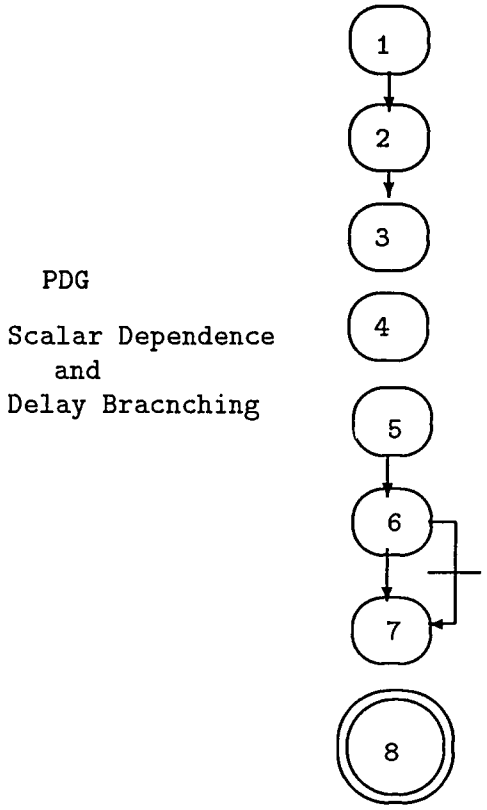
Figure 3.4: PDG Scalar Dependence

3.3.2 Delay Branching Dependence

Definition 3.1 A delay branch exits $\exists\ (I_i = \text{branch}\ ) \wedge (I_i \notin (I_j \delta I_i))$

Previously we defined a *basic block* of instructions as a straight line sequence of instructions within which the existence of a branch instruction may appear only as the last instruction in the block. If this branch instruction is void of any dependence within the *basic block* it can be used in *resource elimination.* Usually this type of branch is an *unconditional branch* or a *call statement.*

At this point the compiler marks the *delay branch* node of the PDG. The following example shows the delay branching instruction with a double circle using the code given in figure 3.4.

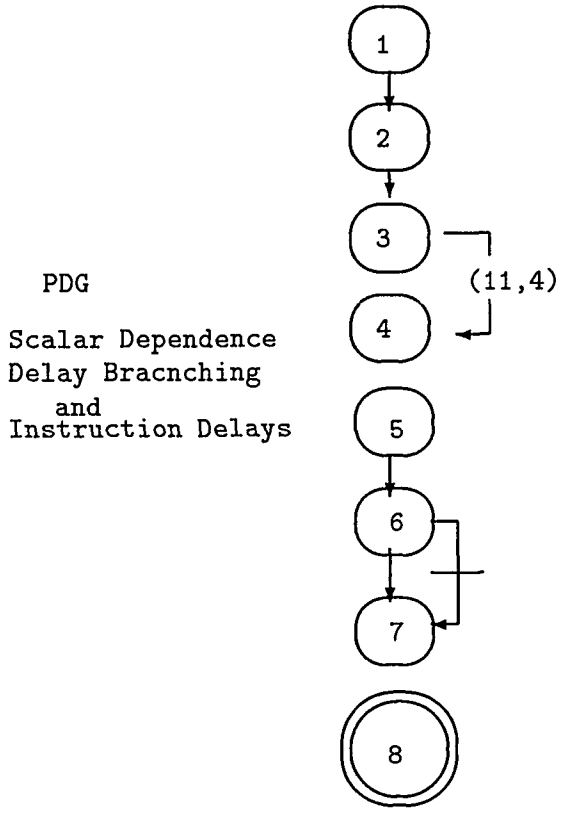


3.3.3 Instruction Timing Dependence

In this particular case the analysis focuses on the the type of architectural bottlenecks that are created by a contention for some hardware resource, such as a single functional unit.

Each arc in the PDG will be assigned a weight and a resource identification number. The weight will represent the instruction found in that statement along with the resource id number.

Using the code given in figure 3.4 and assuming the floating point unit id=11 and its delay = 4 cycles statement our PDG makes another transformation.



3.4 Resource Conflict Removal Techniques

This section will explain optimization techniques that eliminate *resource conflicts* within the pipeline. For each instruction the IN,OUT sets, DDG, and an illustration of a 4-stage pipeline will be included. Dotted lines within each pipeline will reference some data dependence that creates a *resource conflict.*

3.4.1 Code reordering

Definition 3.1 Code reordering exits if $\exists\ I_i \notin (\ (I_i \delta^* I_j) \wedge (I_j \delta^* I_i)\ )$.

If an instruction exists which has no dependencies within the *basic block*, that instruction may be placed lexically anyhere within the *basic block.* Changing the lexical sequence of a *basic instruction block* may break the dependence and remove a *resource conflict.* Nodes in a DDG which have no incoming or outgoing arcs are prime candidates for code reordering. The absence of all arcs defines total independence of an instruction, and allows the compiler the option of reordering the instruction stream within the *basic block.*

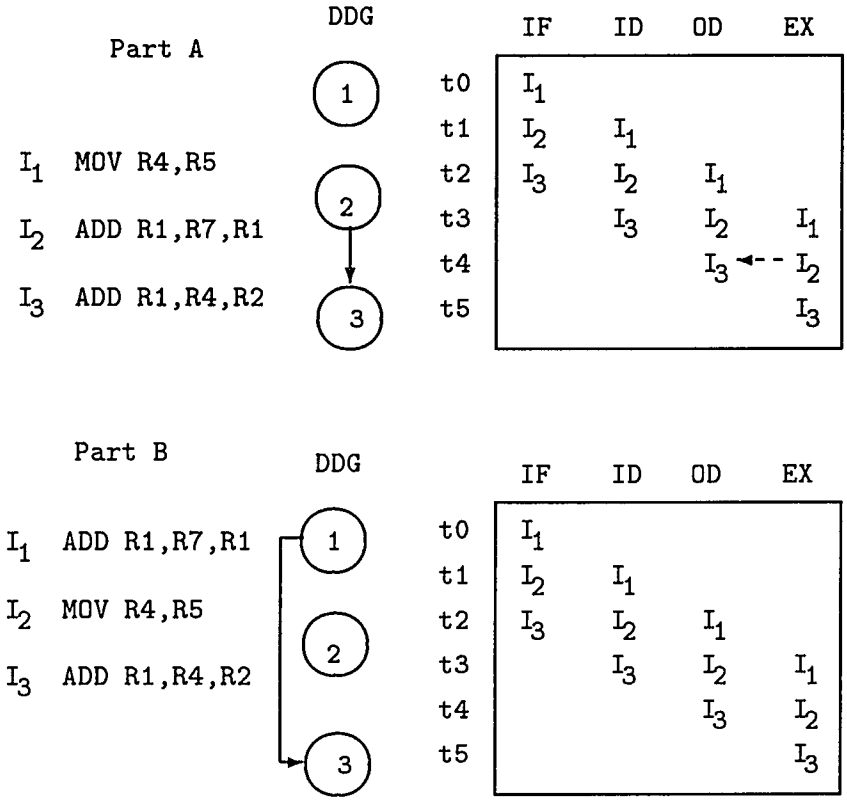


Figure 3.5: Conflict removal by code reordering

In part A of figure 3.5, instruction 1 is void of any dependence whereas instructions 2 and 3 form an adjacent dependence. Instruction 2 is attempting a write of R1, while instruction 3 is attempting a read of R1. In some pipelines, such as our example, an adjacent dependence may cause a *resource conflict.*

In figure 3.5 part B, the elimination of the resource conflict was accomplished by separating instructions 2 and 3 with instruction 1. Notice, this optimization does not eliminate the original dependence but does eliminate the *resource conflict.*

3.4.2 Delay branching

Definition 3.1 A delay branch exits if $\exists$ I_i = branch $\wedge$ $I_i \notin (I_j \delta I_j)$.

Previously we defined a *basic block* of instructions as a straight line sequence of instructions within which the existence of a branch instruction may appear only as the last instruction in the block. If this branch instruction is void of any dependence within the *basic block* it can be used in *resource elimination.* Usually this type of branch is an *unconditional branch* or a *call statement.*

A natural characteristic of a pipeline is the ability to prefetch one instruction, while a previous instruction is being executed. When the executed instruction is a successful branch or an unconditional one, the prefetched instructions must be *flushed* from the pipeline. When *flushing* occurs in a

pipeline the *flushed* instructions add to the wasted memory access time, and the computing time for a task increases. The most common method of dealing with this problem is called *delay branching.*

Figure 3.6 is an example of delay branching in a pipeline which has the ability to hold four instructions.

	Part A	Part B
I_1	LOAD [R2+R4],R1	JMP 105
I_2	ADD R4,R5,R3	LOAD [R2+R4],R1
1_3	JMP 105	ADD R4,R5,R3
I_4	STORE R3,[R5+R7]	STORE R3,[R5+R7]
I_5	SUB R5,R2,R9	SUB R5,R2,R9
I_6	MOV R2,R4	MOV R2,R4

Figure 3.6: Delay branching

Notice, in part A of figure 3.6 , that the execution of $\mathbf{I_3}$, the JMP instruction, will demand that the next instruction to execute is $\mathbf{I_{105}}$. At the same time, instructions $\mathbf{I_4}, \mathbf{I_5}$ and $\mathbf{I_6}$ are being processed by the pipeline. Since the next instruction to execute after the jump will be $\mathbf{I_{105}}$ the pipeline would halt, $\mathbf{I_4}, \mathbf{I_5}$ and $\mathbf{I_6}$ would be flushed, and $\mathbf{I_{105}}$ would enter the pipeline. To eliminate this problem *delay branching* is implemented. Notice in part B of figure 3.6, the code is reordered and the JMP instruction become $\mathbf{I_1}$. When $\mathbf{I_1}$, the JMP, is executed it will demand that the next instruction to enter the pipeline is $\mathbf{I_{105}}$ and that will be the case in *delay branching..*

Delay branching can be utilized in the elimination of *resource conflicts* by analyzing the DDG, and performing code reordering.

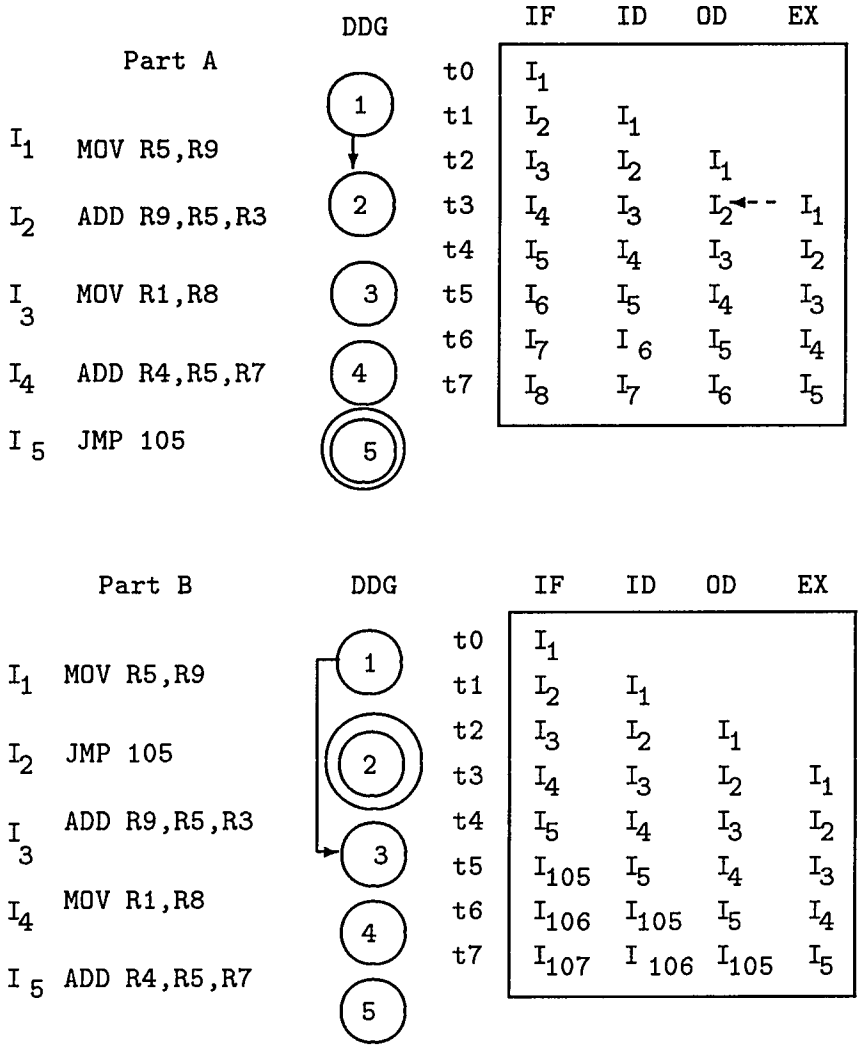


Figure 3.7: Conflict elimination by delay branching

3.4.3 NOPing

The NOPing optimization technique will be the last effort in the *resource conflict* elimination. The insertion of a NOP instruction is a well known technique implemented in RISC compilers [Tabak, 1987]. By having the compiler insert a NOP instruction eliminates the need for a pipeline halt by hardware. Although this adds an extra instruction to the stream, the need for a hardware solution, which is more costly, is eliminated.

Again we will incorporate this technique in the elimination of *resource conflicts.*

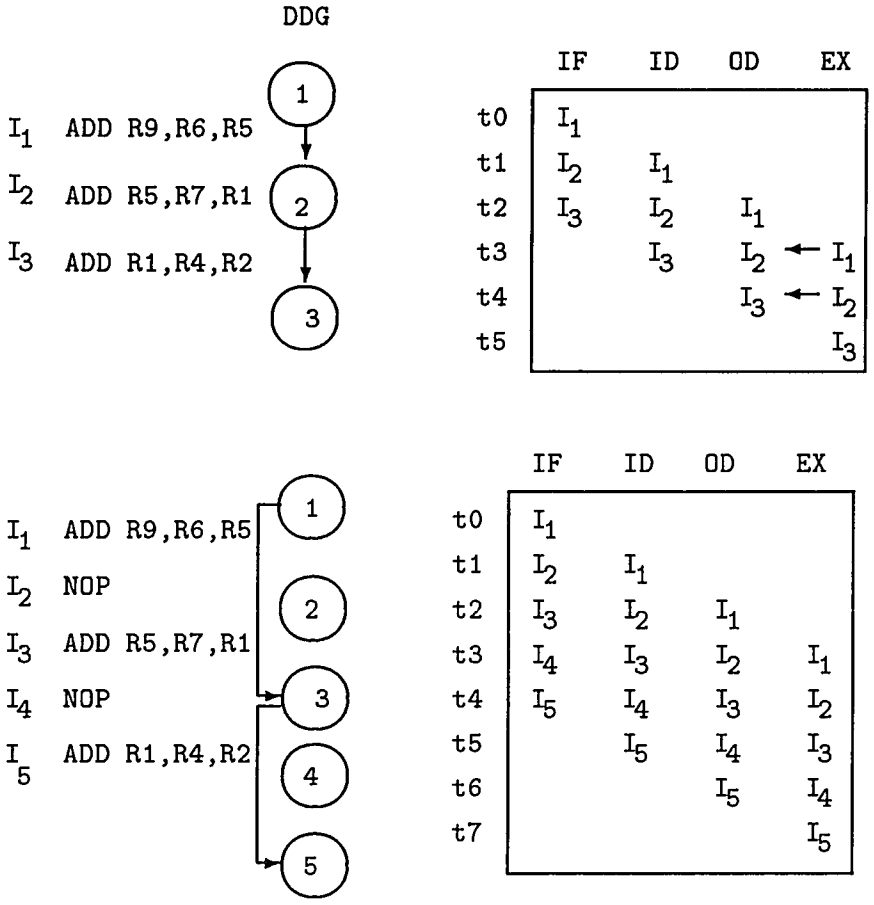


Figure 3.8: Conflict elimination by NOPing

3.5 Transformation Process

The overall approach can be seen in the following steps:

1: Loop Transformations

For all loops do

begin

- compute direction and distance vectors
- create ISDG.

if (direction vectors = undefined) *then*

- do nothing;

else

if (loops are tightly nested) and

(loop limits of inner and outer loops are invariant) and

(there is no dependence relation $\mathbf{S_i}\delta^{*}_{(<,>)}\mathbf{S_j}$) *then*

- perform loop interchange

if (loop is a single loop or the most inner nested loop) and

(cycles exist in ISDG) and

(there is no dependence relation $\mathbf{S_i}\delta^{*}_{(=)}\mathbf{S_j}$) *then*

- perform loop code motion;

end

2: Basic Block Transformation

For all basic blocks do

begin

- Compute **IN** and **OUT** sets.
- Create DDG.
- Transform DDG into PDG by adding delay branching dependence and instruction timing dependence.
- Pick the first node which has an outgoing arc data dependence arc but no incoming data dependence arc.
  Set **WINDOW** = first node and let the **WINDOW** size be N. Where N is the number of active instructions allowed in the pipeline at anyone time.
  if that node does not exist *then*
  - goto 3;
- Give each node in the PDG a weight = to the number of children it has.
- Initially, let **CANDIDATE-SET** represent the instructions that have no incoming arcs, do not include the delay branch instruction.

while **WINDOW** $\neq$ **empty** do

begin

while **WINDOW has dependence** do

begin

if (**CANDIDATE-SET** = empty)

• Insert NOP before the lower node of

the dependence.

else

• Remove a node from the **CANDIDATE-SET**

not in this dependence with the largest weight.

Add any of its successors to the **CANDIDATE-SET**

which now have no incoming arcs.

• Insert that node before the lower

node of the dependence.

• Update **WINDOW** with the new

sequence of N instructions.

end

• Mark top node of **WINDOW USED** and if it

is a member of the **CANDIDATE-SET** remove it.

• Move **WINDOW** down one level in PDG.

end

3: Instruction Timing Transformation

For possible instruction timing dependence do

begin

- Update PDG with instruction dependence.

while (instruction dependence arc > 0) do

begin

if (**CANDIDATE-SET** = empty)

- Insert NOP before the lower node of

the dependence

and decrement instruction dependence arc by 1.

else

- Remove a node from the **CANDIDATE-SET**

which now have no incoming arcs or outgoing arcs.

- Insert that node before the lower

node of the dependence.

and decrement instruction dependence arc by 1.

end

end

4: Delay Branch Transformation

- Find the $(N+1)^{th}$ instruction from the bottom of the PDG, where N is the number of stages in the pipeline. Label that node **D_B**.

if (**D_B** = NOP) then

- replace the **D_B** with the delay branch instruction.

else

- Insert delay branch instruction after **D_B**.

3.6 An Example of the Transformation Process.

The segment of code found in figure 3.9 will help us illustrate a code transformation sequence during optimization. The reordering is targeted at pipeline $\mathbf{P_{4,4}}$ where all hazards occur in adjacent slots of the pipeline. That is between I_1 and I_2 then I_2 and I_3 ... I_{n-1} and I_n. Also, in this example the only instruction with a timing delay will be the FDIV. Where the floating point unit has id=11 and a stall of 4 cycles. Stalls will appear as blank lines in the pipeline.

3.6.1 Transformation Example

I_1	FDIV R3,R14,R1	$IN(1) = \{R3, R14\}$	$OUT(1) = \{R1\}$
I_2	ADD R1,R2,R12	$IN(2) = \{R1, R2\}$	$OUT(2) = \{R12\}$
I_3	FDIV R9,R5,R4	$IN(3) = \{R9, R2\}$	$OUT(3) = \{R4\}$
I_4	ADD R9,R4,R6	$IN(4) = \{R9, R4\}$	$OUT(4) = \{R6\}$
I_5	ADD R9,R2,R0	$IN(5) = \{R9, R2\}$	$OUT(5) = \{R0\}$
I_6	LOAD [R9+R0],R7	$IN(6) = \{R9, R0, mem[R9 + R0]\}$	$OUT(6) = \{R7\}$
I_7	STORE R10,[R11]	$IN(7) = \{R10, R11\}$	$OUT(7) = \{mem[R11$
I_8	JMP 137		

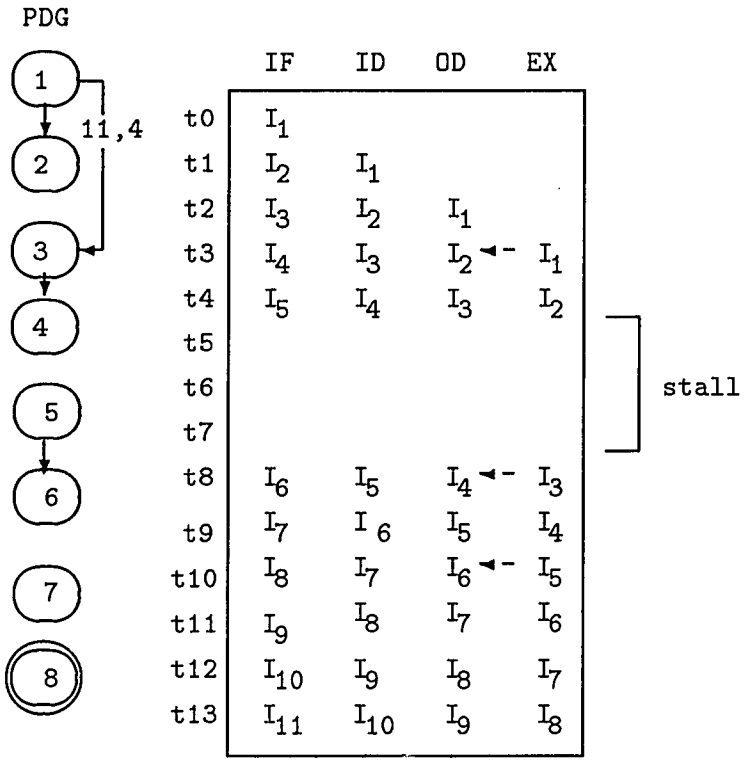


Figure 3.9: Example code before transformation.

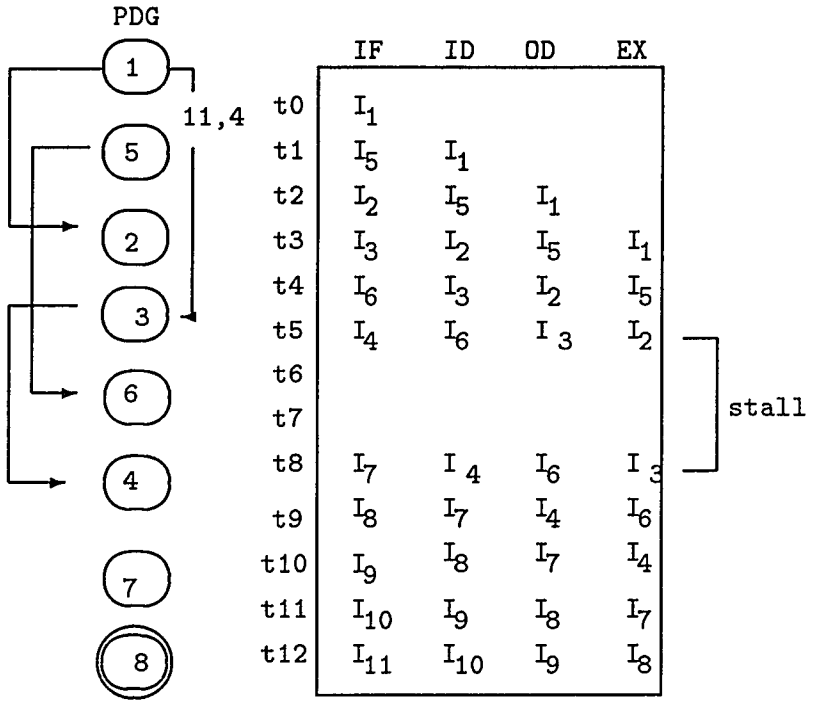


Figure 3.10: Example Code after basic block transformation..

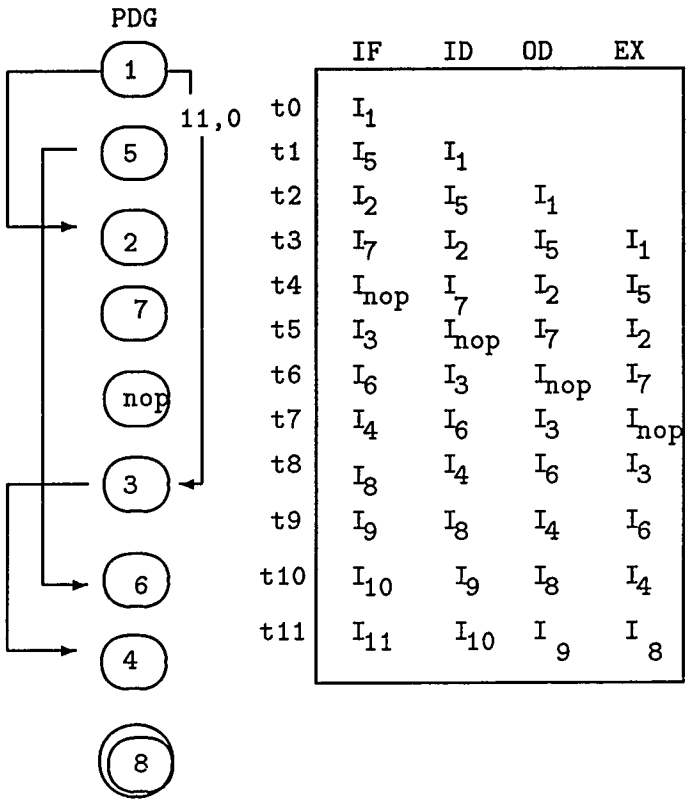


Figure 3.11: Example code after instruction timimg transformation..

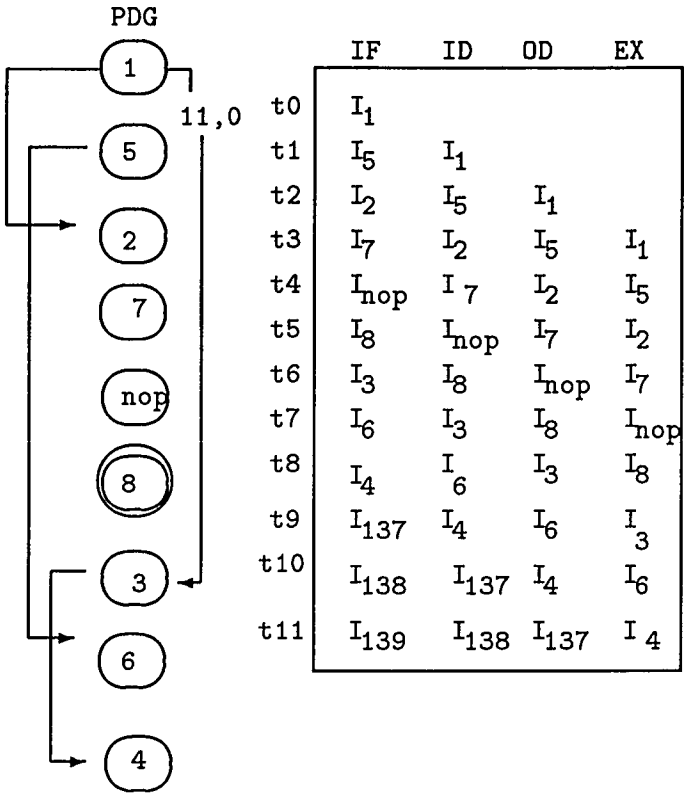


Figure 3.12: Example code after branch delay transformation..

The rescheduled code show in figure 3.12 is now ready for that specific pipeline. All possibel conflicts and stalls have been removed.

3.7 Order of Optimizations ...?

At this time the order in which optimizations should be performed is an open problem. This section has been added to show that some well know

optimization techniques lend themself to our technique. This chapter will explain techniques that eliminate *resource conflicts* within the pipeline. For each instruction the IN,OUT sets, DDG, and an illustration of a 4-stage pipeline will be included. Dotted lines within each pipeline will reference some data dependence that creates a *resource conflict.*

3.7.1 Folding

Definition 3.1 Folding exits if $\exists$ $\mathbf{I_i}$ **var** $:=$ **con** $\wedge$ $\exists$ $\mathbf{I_j} \in (\mathbf{I_i} \delta \mathbf{I_j})$.

Thus, if an instruction assigns a constant to a variable, and there exists a flow dependence from that variable to a future instruction, a substitution of that constant value for any future undefined *use* of that variable is allowed. To illustrate, if an instruction has the form of $R_i := R_j$, we substitute the value of R_j for any future undefined *uses* of R_i.

In part A of figure 3.13, the definition for folding holds since I_1 **R3** $:=$ **var** $\wedge$ $(\mathbf{I_1} \delta \mathbf{I_2})$. When folding is applied, as shown in part B of figure 3.13, the dependence is broken and parallelism is created between $\mathbf{I_1}$ and $\mathbf{I_2}$. The transformation substituted the value of R3 from instruction 1, for R1 in instruction 2.

3.7.2 Dead code elimination

Definition 3.1 Dead code exits if

$\exists$ $\mathbf{I_j}$ **such that** $(\mathbf{I_j} \in (\ \mathbf{I_i}\ \delta\ \mathbf{I_j}\ ) \vee (\ \mathbf{I_j} = \mathbf{I_1}\ )) \wedge \nexists\ \mathbf{I_k} \in (\ \mathbf{I_j}\ \delta\ \mathbf{I_k})$.

Thus, if an instruction is dependent on a previous instruction or it is the first instruction of the *basic block*, and there is no flow to a future instruction before it is redefined, it is labeled useless and can be eliminated.

As before, when an bistruction has the form of $R_i := R_j$, we substitute the value of R_j for any future nondefined *uses* of R_i. The instruction $R_i := R_j$ is then identified as dead code and is removed from the instruction stream.

In part B of figure 3.13, the definition for dead code elimination holds since $(\mathbf{I_j} = \mathbf{I_1}) \wedge \nexists \mathbf{I_k} \in (\mathbf{I_1} \ \delta \ \mathbf{I_k})$.

Notice that this transformation does not effect the pipeline, although it will improve the overall performance of the computation and is traditionally implemented after a *folding optimization* [Sethi and Ullman, 1986].

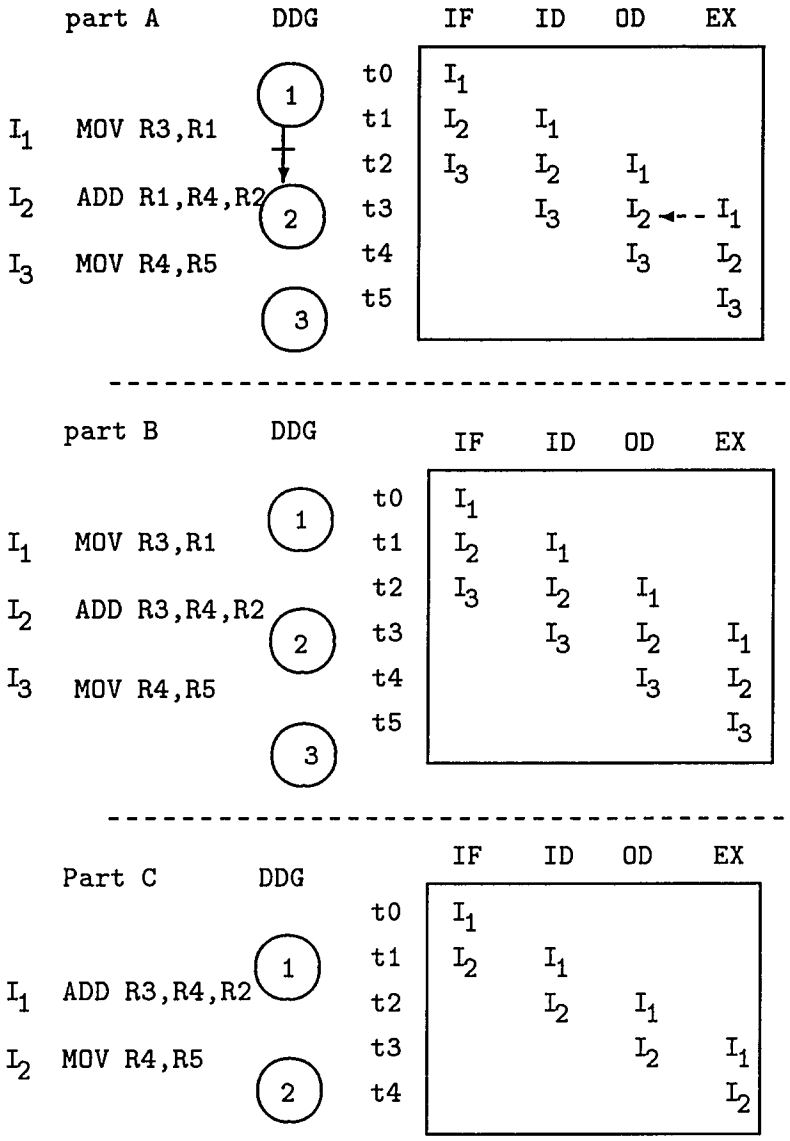


Figure 3.13: Conflict elimination by folding and dead code removal

Chapter 4

Conclusions and Further Research

4.1 Conclusion

This work has presented a software approach to code reorganization which maximizes the parallel performance of an instruction pipeline. We accomplished this by focusing the approach on the elimination of specific *arcs* in the dependence graph where an arc can represent a type of *pipeline hazard* known as the *resource conflict.*

Summarizing the approach, the compiler computes direction and distance vectors for each loop structure . It then creates an ISDG and examines the loop for possible optimizations which break loop iteration dependence. After loop dependence is broken the **IN() OUT()** sets for each basic statement block are created. By analyzing the **IN() OUT()** sets a PDG is created. When inspecting the dependence graph the compiler is able to detect the dependencies that cause *resource conflicts.* Optimizations are then carried

out on the instruction block resulting in a revised version of the original code void of *resource conflicts.* A complete example with transformations within a instruction block is shown with the corresponding PDG and **IN() OUT()** sets.

The optimizations include : loop interchange, loop statement motion, folding, dead code elimination, code reordering, NOPing, and delay branching.

4.2 Further Research

- Evaluate the general effectiveness of this approach by simulation.
- Evaluate the performance by implementing this approach on different pipelined systems.
- Investigate the interactions between the optimizations to determine whether the order in which the optimizations are applied effects the overall performance of the pipeline.

Bibliography

[Ackerman, 1981] Ackerman, W. B. (1981). Data flow languages. *Tutorial on Parallel Processing*, pages 335–343.

[Allen and Cocke, 1972] Allen, F. and Cocke, J. (1972). *A Catalogue of Optimizing Transformations.* Prentice-Hall, Englewood Cliffs, N.J.

[Allen, 1986] Allen, F. E. (1986). Compiling for parallelism. *Proceedings of the 1986 IBM Europe Institute Seminar on Parallel Computing.*

[Allen, 1983] Allen, J. (1983). *Dependence Analysis for Subscripted Variables and It's Application to Program Transformation.* PhD thesis, Rice University (UMI 83-14916).

[Allen and Kennedy, 1982] Allen, J. and Kennedy, K. (1982). Pfc: A program to convert fortran to parallel form. Technical report, Rice University. Technical Report MASC-TR82-6.

[Almasi and Gottlieb, 1989] Almasi, G. S. and Gottlieb, A. (1989). *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc, Redwood City, California 94065.

[Aray, 1985] Aray, S. (1985). An optimal instruction scheduling model for a class of vector processors. *IEEE Transactions on Computers*, 34(11).

[Banerjee, 1976] Banerjee, U. (1976). *Data Dependence in Ordinary Programs.* PhD thesis, University of Illinois, Urbana-Champaign.

[Banerjee, 1979] Banerjee, U. (1979). *Speedup of Ordinary Programs.* PhD thesis, University of Illinois, Urbana-Champaign.

[Beatty, 1972] Beatty, J. (1972). An axiomatic approach to code optimization for expressions. *ACM*, 19(4):613–640.

[Beatty, 1974] Beatty, J. (1974). Register assignment algorithm for generation of highly optimized object code. *IBM J. res. Dev.*, 18(1):20–39.

[Berstien, 1988] Berstien, D. (1988). An improved approximation algorithm for scheduling pipelined machines. *Proceedings of the International Conference on Parallel Processing.*

[Broy, 1986] Broy, M. (1986). *Control Flow and Data Flow : Concepts of Distributed Programming.* Springer–Verlag New York.

[Charles and Ferrante, 1987] Charles, F. M. B. P. and Ferrante, J. (1987). An overview of the ptran analysis system for multiprocessing. *Lecture Notes in Computer Science*, 297:195–211.

[Chow and Rudmik, 1982] Chow, A. and Rudmik, A. (1982). The design of a data flow analyzer. *ACM Proceedings of the SIGPLAN 82 Symposium on Compiler Construction*, pages 106–113.

[Chow, 1989] Chow, P. (1989). *The MIPS-X RISC Microprocessor.* Kluwer Academic Publishers.

[Cooper, 1983] Cooper, K. (1983). *Interprocedural Data Flow Analysis in a Programming Environment.* PhD thesis, Rice University.

[Cooper and Kennedy, 1984] Cooper, K. D. and Kennedy, K. (1984). Efficient computation of flow insensitive interprocedural summary information. *ACM Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, 19(6):247–258.

[Cytron, 1986] Cytron, R. (1986). On the application of dependence analysis and restructuring techniques to parallel and functional languages. *Proceedings of the 1986 IBM Europe Institute Seminar on Parallel Computing.*

[Cytron and Ferrante, 1987] Cytron, R. and Ferrante, J. (1987). What's in a name? the value of renaming for parallelism detection and storage allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27.

[D. Gellernter and Padua, 1990] D. Gellernter, A. N. and Padua, D. (1990). *Languages and Compilers for Parallel Computing.* MIT Press, Cambride, Massachusetts.

[D. Kuck, 1980] D. Kuck (1980). *High-Speed Machines and Their Compilers*, Proceedings of the CREST Parallel Processing Systems Cource, Cambridge. Cambridge University Press.

[Dasgupta, 1989a] Dasgupta, S. (1989a). *Computer Architecture volume 1.* John Wiley and Sons, New York.

[Dasgupta, 1989b] Dasgupta, S. (1989b). *Computer Architecture volume 2.* John Wiley and Sons, New York.

[D.Bernstien and Gertner, 1989a] D.Bernstien and Gertner, I. (1989a). Scheduling expressions on a pipelined with a maximal delay of one cyle. *ACM Transactions on Programming Languages and Systems*, 11(1).

[D.Bernstien and Gertner, 1989b] D.Bernstien, M. R. and Gertner, I. (1989b). Approximation algorithms for scheduling arithmetic expressions on pipelined machines. *Journal of Algorithms*, 10(1).

[de Bakker et al., 1986] de Bakker, J., Kok, J., Meyer, J., Olderog, E., and Zucker, J. (1986). Contrasting themes in the semantics of imperative concurrency. *Lecture Notes in Computer Science*, 224:51–121.

[D'Hollander and Opsommer, 1987] D'Hollander, E. H. and Opsommer, J. (1987). Implementation of an automatic program partitioner on a homgeneous mutiprocessor. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 517–519.

[Dinning and Schonberg, 1990] Dinning, A. and Schonberg, E. (1990). An empirical comparison of monitoring algorithms for access anomaly detection. *Second ACM SIGPLAN on Principles and Practice of Parallel Programming PPOPP*, 25(3):1–10.

[Ellis, 1985] Ellis, J. (1985). *Bulldog: A Compiler for VLIW Architectures.* MIT Press.

[Fisher, 1987] Fisher, J. (1987). *Wide Instruction Word Architectures: Solving the Supercomputer Software Problem.* Elsevier Science Publishers B.V., 52 Vanderbilt Ave. New York N.Y. 10017 U.S.A.

[Fisher et al., 1984] Fisher, J. A., Ellis, J. R., Ruttenberg, J. C., and Nicolau, A. (1984). Parallel processing: A smart compiler and a dumb machine. *ACM Proceedings of the SIGPLAN 84 Symposium on Compiler Construction Montreal, Canada*, 19(6):37–47.

[Gibbons and Muchnick, 1986] Gibbons, P. and Muchnick, S. (1986). Efficient instruction scheduling for pipelined architectures. *ACM Proceedings of the SIGPLAN 86 Symposium on Compiler Construction Palo Alto.*

[Griffin, 1954] Griffin, H. (1954). *Elementary Theory of Numbers.* McGraw-Hill.

[Gurd and Bohm, 1986] Gurd, J. and Bohm, W. (1986). Implict parallel processing:sisal on the manchester dataflow computer. *Proceedings of the 1986 IBM Europe Institute Seminar on Parallel Computing*, pages 179–204.

[Hecht, 1977] Hecht, M. S. (1977). *Flow Analysis of Computer Programs.* Prentice–Hall, Inc.

[Hendren and Nicolau, 1990] Hendren, I. and Nicolau, A. (1990). Parallelizing programs with recursive data structures. *Parallel and Distributec Systems*, 1(1):35–47.

[Hennessy and Patterson, 1990] Hennessy, J. and Patterson, D. (1990). *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers,Inc., San Mateo, California.

[J.Banning, 1978] J.Banning (1978). *A Method for Determining the Side Effects of Procedure Calls.* PhD thesis, Standard, University.

[Jeane Ferrante and Warren, 1987] Jeane Ferrante, K. O. and Warren, J. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

[Jesshope, 1986] Jesshope, C. (1986). Building and binding systems wth transputers. *Proceedings of the 1986 IBM Europe Institute Seminar on Parallel Computing.*

[J.Hennesey and Gross, 1983] J.Hennesey and Gross, T. (1983). Postpass code optimization of pipeline constraits. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448.

[Kennedy, 1984] Kennedy, J. A. K. (1984). Automatic loop interchange. *ACM Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, 19(6):247–258.

[Kennedy and Subhlok, 1988] Kennedy, V. B. D. B. D. C. K. and Subhlok, J. (1988). Ptool: A system for static analysis of parallel programs. Technical report, Rice University. Technical Report COMP-TR88-71.

[Kirch, 1974] Kirch, A. (1974). *Elementary Number Theory.* Intext.

[K.Kennedy, 1975] K.Kennedy (1975). Use-definiton chains with aplications. Technical report, Rice University. Technical Report 476-093-9.

[Krishnamurthy, 1989] Krishnamurthy, E. (1989). *Parallel Processing : Principles and Practice.* Addison–Wesley.

[Kuck et al., 1981] Kuck, D., Budnik, P., Chen, S., Lawrie, D., Towle, R., and Strebent, R. (1981). Parallelism in ordinary fortran programs. *Tutorial on Parallel Processing*, pages 346–362.

[Lowry and Medlock, 1969] Lowry, E. and Medlock, C. (1969). Object code optimization. *ACM*, 12(1):13–22.

[Mickunas and Schell, 1978] Mickunas, M. and Schell, M. (1978). Parallel compilation in a multiprocessor environment. *ACM Proceedings 1978 Annual Conference*, pages 241–246.

[Moor, 1982] Moor, I. (1982). An applicative compiler for a parallel machine. *ACM Proceedings of the SIGPLAN 82 Symposium on Compiler Construction Boston Massachusetts*, 17(6):284–293.

[Muchnick and Jones, 1981] Muchnick, S. and Jones, N. (1981). *Program Flow Analysis : Theory and Applications.* Elsevier Scientific Publishing Company.

[N.D.Jones and S.Muchnick, 1982] N.D.Jones and S.Muchnick (1982). A flexible to interprocedural data flow analysis and programs with recursive data structures. *9th ACM Symposium on the Priciples of Programming Languages*, pages 66–74.

[Padua and Wolfe, 1986] Padua, D. and Wolfe, M. (1986). Advanced compiler optimizations for supercompurers. *Communications of the ACM*, 29(12):1184–1201.

[Pfeiffer and Reps, 1989] Pfeiffer, S. H. P. and Reps, T. (1989). Dependence analysis for pointer variables. *Proc. SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 28–40.

[Polychronopoulos, 1987a] Polychronopoulos, C. (1987a). On advanced compiler optimizations for parallel computers. *Proceddings of the International Confercnce on Supercomputing.*

[Polychronopoulos, 1987b] Polychronopoulos, C. D. (1987b). Automatic restructuring of fortran programs for parallel execution. *Lecture Notes in Computer Science*, 295:107–130.

[Polychronopoulos, 1987c] Polychronopoulos, C. D. (1987c). Loop coalescing: A compiler transformation for parallel machines. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 235–242.

[Remi Triolet, 1986] Remi Triolet, Francois, P. F. (1986). Direct parallelization of call statements. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction Palo Alto*, 21(7):176–185.

[Sarkar and Hennessy, 1986] Sarkar, V. and Hennessy, J. (1986). Compile-time partioning and scheduling of parallel programs. *ACM Proceedings of the SIGPLAN 86 Symposium on Compiler Construction Palo Alto*, pages 17–26.

[Schneck, 1975] Schneck, P. (1975). Movement of implicit parallel and vector expressions out of program loops. *SIGPLAN Notices*, 10(3):103–106.

[Sethi et al., 1970] Sethi, Ravi, and Ullman, J. (1970). The generation of optimal code for arithmetic expressions. *ACM*, 17(4):715–728.

[Sethi and Ullman, 1986] Sethi, A. A. R. and Ullman, J. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts.

[Sharp, 1985] Sharp, J. A. (1985). *Data Flow Computing.* Ellis Horwood Limited.

[Tabak, 1987] Tabak, D. (1987). *RISC Architecture.* Research Studies Press LTD., England.

[Thomasset and Eisenbeis, 1986] Thomasset, A. L. F. and Eisenbeis, C. (1986). Automatic detection of parallelism in scientific programs with application to array-processors. *Proceedings of the 1986 IBM Europe Institute Seminar on Parallel Computing.*

[Tiemann, 1989] Tiemann, M. (1989). Pfc: The gnu instruction scheduler. Technical report, Stanford University. CS343 cource report.

[Wolfe, 1982] Wolfe, M. (1982). *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, Urbana-Champaign.

[Yew and Zhu, 1990] Yew, Z. L. P.-C. and Zhu, C.-Q. (1990). An efficient data dependence analsis for parallelizing compilers. *Parallel and Distributed Systems*, 1(1):26–34.