The exploitation of pipeline parallelism by compile time dataflow analysis

Joseph Michael Lombardo

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/149

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
The exploitation of pipeline parallelism by compile time dataflow analysis

Lombardo, Joseph Michael, M.S.

University of Nevada, Las Vegas, 1991
THE EXPLOITATION OF PIPELINE
PARALLELISM BY COMPILERTIME 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 Datta
Examining Committee Member, Ajoy Datta Ph.D.

Rohan Dalpatadu
Graduate Faculty Representative, Rohan Dalpatadu Ph.D.

Ronald Smith
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

1 Introduction ................................................. 1

2 Overview ..................................................... 3
   2.1 Instruction Timing and Stalls .......................... 3
      2.1.1 The MIPS R2000 ............................... 7
      2.1.2 The Stanford MIPS ............................. 8
      2.1.3 The IBM 801 ..................................... 9
      2.1.4 The GMU Microcoded RISC-MIRIS ........... 9
      2.1.5 The ACORN RISC Machine ................... 9
   2.2 Data Reference and Stalls ............................ 10
   2.3 Branching and Stalls .................................. 11
   2.4 Previous Work ......................................... 13
   2.5 Survey of Dataflow Analysis .......................... 15
   2.6 Graphs ................................................... 19
      2.6.1 Data Dependence Graphs, DDG ................ 19
      2.6.2 Iteration Space Dependence, ISDG ............ 20
3 Instruction Pipeline Optimization via Dataflow Analysis 27

3.1 Pipeline Dependence .......................... 28
  3.1.1 Flow Pipeline Dependence .............. 28
  3.1.2 Anti Pipeline Dependence .............. 29
  3.1.3 Output Pipeline Dependence .......... 30

3.2 Pipeline Iteration Space Dependence .......... 31
  3.2.1 Dependence Free Loops ................. 32
  3.2.2 Loop Interchange ....................... 32
  3.2.3 Loop Code Motion ....................... 34

3.3 Pipeline Dependence Graph, PDG .......... 36
  3.3.1 Data Dependence Graph ................. 36
  3.3.2 Delay Branching Dependence .......... 39
  3.3.3 Instruction Timing Dependence ....... 40

3.4 Resource Conflict Removal Techniques .......... 42
  3.4.1 Code reordering ....................... 42
  3.4.2 Delay branching ....................... 44
  3.4.3 NOPing ................................ 47

3.5 Transformation Process ....................... 49

3.6 An Example of the Transformation Process .......... 53
  3.6.1 Transformation Example ............... 54

3.7 Order of Optimizations ? .................... 57
3.7.1 Folding .......................................................... 58
3.7.2 Dead code elimination ................................. 58

4 Conclusions and Further Research ........................ 61

4.1 Conclusion ......................................................... 61
4.2 Further Research .................................................... 62
List of Figures

2.1  An example of a pipeline. ......................................................... 5
2.2  A pipeline hazard and stall at a cost of one machine cycle. .... 6
2.3  A pipeline hazard and stall at a cost of 19 machine cycles. .. 6
2.4  Resource conflict on R1 ............................................................ 11
2.5  A pipeline stall due to branching. ............................................. 12
2.6  Data Dependent Graph, DDG ..................................................... 20

3.1  Flow Pipeline Dependence ....................................................... 28
3.2  Anti Pipeline Dependence ........................................................ 29
3.3  Output Pipeline Dependence .................................................... 31
3.4  PDG Scalar Dependence .......................................................... 38
3.5  Conflict removal by code reordering ........................................ 43
3.6  Delay branching ................................................................. 45
3.7  Conflict elimination by delay branching ................................. 46
3.8  Conflict elimination by NOPing .............................................. 48
3.9  Example code before transformation ....................................... 55
3.10 Example Code after basic block transformation ....................... 55
3.11 Example code after instruction timing transformation  . . . 56
3.12 Example code after branch delay transformation  . . . . . . . . 57
3.13 Conflict elimination by folding and dead code removal  . . . 60
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 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 \( t \), 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 of 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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>22</th>
<th>23</th>
<th>...</th>
<th>41</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDIV</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 Stanford 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 \quad (R1 := R4)
\text{i + 1 ADD R1,R2,R2 \quad (R2 := R1 + R2)}

\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
\text{i} & \text{i+1} & \text{Machine Cycles} \\
\hline
\text{} & \text{} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
\text{i} & \text{i+1} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{SR} \\
\text{} & \text{} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{SR} \\
\hline
\end{tabular}

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 \textit{flushed} from the pipeline. When \textit{flushing} occurs in a pipeline the \textit{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.
Notice in figure 2.5 that the pipeline was filled with five instructions.
where 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 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. Whereas 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.

¹An 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 creates 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.
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.

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

Consider the next few examples:

**Example 1**

\[ I_1 \bar{d}_{(>)} I_2 \]

\[
\begin{align*}
\text{DO } i \text{ } = \text{ } 1 \text{ to } N & \quad \text{equation: } i=j-1 \\
I_1: \quad A(i) &= \\
I_2: \quad =A(i-1) & \quad \text{possible solution. } i=1 \ j=2 \\
\text{ENDDO} & \quad \text{distance: } j-i=1 \\
& \quad \text{direction: } j-i>0
\end{align*}
\]

**Example 2**

\[ I_1 \bar{d}_{(0)} I_2 \]

\[
\begin{align*}
\text{DO } i \text{ } = \text{ } 1 \text{ to } N & \quad \text{equation: } 2i=2j-1 \\
I_1: \quad A(2i) &= \\
I_2: \quad =A(2i-1) & \quad \text{No solutions no dependence} \\
\text{ENDDO} & 
\end{align*}
\]

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

\[ I_1 \delta(>,<) I_2 \]

\[
\begin{align*}
&\text{DO 10 i=1 to N} \\
&\text{DO 20 j=1 to M} \\
&I_1: \quad A(i,j) = \\
&I_2: \quad = A(i-1,j+1)
\end{align*}
\]

\begin{align*}
equation 1: & \quad i = k-1 \\
distance 1: & \quad 1 \quad \text{direction 1: } > \\
equation 2: & \quad j = l+1 \\
distance 2: & \quad -1 \quad \text{direction 2: } <
\end{align*}

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

- \(\text{IB}_n\): ...a basic block of \(n\) instructions \(\text{IB}_n \equiv \{I_1, I_2, \ldots, I_n\}\).

- \(I_i < I_j\): ...when instruction \(I_i\) lexically precedes instruction \(I_j\) in the instruction stream.

- **Instruction Formats** ...memory is access only by the LOAD and
STOR operations.

\[
\begin{align*}
\text{ADD } \text{src1,src2,dst} & \quad \equiv \quad \text{dst} := \text{src1 + src2} \\
\text{SUB } \text{src1,src2,dst} & \quad \equiv \quad \text{dst} := \text{src1 - src2} \\
\text{LOAD } [\text{src1+src2}],\text{dst} & \quad \equiv \quad \text{dst} := \text{memory[src1+src2]} \\
\text{MOV } \text{src1,dst} & \quad \equiv \quad \text{dst} := \text{src1} \\
\text{STORE } \text{src1,dst} & \quad \equiv \quad \text{memory[dst]} := \text{src1} \\
\text{JMP } \text{dst} & \quad \equiv \quad \text{PC} := \text{dst} \\
\text{INC } \text{src1} & \quad \equiv \quad \text{src1} := \text{src1 + 1}
\end{align*}
\]

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

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

- **Pipeline Formats** \ldots \text{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:

- **S₁**:
- **S₂**:
- **Sₙ**:  
  ::
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 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-

---

1A 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 $I_j \delta I_j : I_j$ is flow dependent on $I_i$. $I_i$ must store its results before $I_j$ is allowed to execute. Flow dependence exists in pipeline $P_{s,a}$ iff

$$(|i - j| < a) \land (I_i < I_j) \land (\text{OUT}(I_i) \cap \text{IN}(I_j) \neq \emptyset)$$

$i$ \hspace{1cm} MOV R4,R1 \hspace{1cm} (R1 := R4)

$i + 1$ \hspace{1cm} ADD R1,R2,R2 \hspace{1cm} (R2 := R1 + R2)

Figure 3.1: Flow Pipeline Dependence
A flow dependence can be seen in figure 3.1. At time 3, instruction $i + 1$ will access R1 for a read, while instruction $i$ will not write that value until time 5. Notice the formal condition holds since

$$(|i - j| < a) \land (i < i + 1) \land (\text{OUT}(i) = \{R1\} \cap \text{IN}(i + 1)\{R1\} = \{R1\} \neq \emptyset)$$

### 3.1.2 Anti Pipeline Dependence

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

$$(|i - j| < a) \land (I_i < I_j) \land (\text{IN}(I_i) \cap \text{OUT}(I_j)) \neq \emptyset$$

$i$  LOAD mem[R1],R4  $(R4 := \text{mem}[R1])$

$i + 1$  STORE R6,mem[R1]  $(\text{mem}[R1] := R6)$

|  | Machine Cycles |
|---|---|---|---|---|---|---|
|   | 1 | 2 | 3 | 4 | 5 | 6 |
| $i$ | IF | ID | OD | SX | OF |
| $i + 1$ | IF | ID | OD | SX | OF |

Figure 3.2: Anti Pipeline Dependence
An anti pipeline dependence can 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 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

\[(|i - j| < a) \land (i < i) \land (IN(i) = \{R1\} \cap OUT(i + 1)\{R1\} = \{R1\} \neq \emptyset)\]

### 3.1.3 Output Pipeline Dependence

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

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

\[(|i - j| < a) \land (I_i < I_j) \land (OUT(I_i) \cap OUT(I_j) \neq \emptyset)\]

\[
i \quad \text{MOV R4,R1} \quad (R1 := R4)
\]

\[
i + 1 \quad \text{INC R1} \quad (R1 := R1 + 1)
\]

\[
\begin{array}{c|cccccc}
\text{Machine Cycles} & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
i & S_1 & S_2 & S_3 & S_4 & S_5 & \\
i + 1 & S_1 & S_2 & S_3 & S_4 & S_5 & \end{array}
\]

**Figure 3.3**
An output pipeline dependence can 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

\[(|i-j| < a) \land (i < i+1) \land (\text{OUT}(i) = \{\text{var}\} \bigcap \text{OUT}(i+1)\{\text{var}\} = \{\text{var}\} \neq \emptyset)\]

### 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 \( S_i \delta_{[<,>]} S_j \)

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

Example 3.2

**Before Transformation**

```plaintext
DO 10 I = 2 to N
   DO 20 J = 2 to M
      A(I,J) = A(I, J - 1)
   20
10
```
Dependence Equation 1: (=, >)

\[
A(2,2) = A(2,1)
\]
\[
A(2,3) = A(2,2) \quad \text{Flow Dependence}
\]
\[
A(2,4) = A(2,3)
\]

etc.

33
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: \((>,=)\)
\[ A(2,2) = A(2,1) \]
\[ A(3,2) = A(3,1) \quad \text{dependence is broken} \]
\[ A(4,2) = A(4,1) \]
\[ \text{etc.} \]

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 \( S_i \delta_{(=}^i) S_j \)
Example 3.3 shows a loop before and after code motion

Example 3.3

Before Transformation $S_1\delta_{(<)}S_2$

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 $S_1\delta_{(<)}S_2$

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 creates 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 $IB_n$ is a basic block of n instructions, $IB_n \equiv \{I_1, I_2, ..., I_n\}$, and $I_i < I_j$ when instruction $I_i$ lexically precedes instruction $I_j$ in the instruction stream. The compiler extracts the dependence information by using the following definitions;
Flow Dependence

\[(I_i < I_j) \land (\text{OUT}(I_i) \cap \text{IN}(I_j)) \neq \emptyset\]

Anti-Flow Dependence

\[(I_i < I_j) \land (\text{IN}(I_i) \cap \text{OUT}(I_j)) \neq \emptyset\]

Output Dependence

\[(I_i < I_j) \land (\text{OUT}(I_i) \cap \text{OUT}(I_j)) \neq \emptyset\]

The segment of code found in figure 3.4 will help us illustrate scalar data dependence.
| \( I_1 \) | MOV R3,R1 | \( IN(1) = \{ R3 \} \) | \( OUT(1) = \{ R1 \} \) |
| \( I_2 \) | ADD R1,R2,R2 | \( IN(2) = \{ R1, R2 \} \) | \( OUT(2) = \{ R2 \} \) |
| \( I_3 \) | FDIV R9,R2,R4 | \( IN(3) = \{ R9, R2 \} \) | \( OUT(3) = \{ R4 \} \) |
| \( I_4 \) | FDIV R9,R8,R6 | \( IN(4) = \{ R9, R8 \} \) | \( OUT(4) = \{ R6 \} \) |
| \( I_5 \) | 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 R7,[R11] | \( IN(7) = \{ R7, R11 \} \) | \( OUT(7) = \{ mem[R11] \} \) |
| \( I_8 \) | JMP 137 | | |

Figure 3.4: PDG Scalar Dependence
3.3.2 Delay Branching Dependence

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

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 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.

PDG
Scalar Dependence
Delay Branching
and Instruction Delays
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 exists if \( \exists I_i \not\in (I_1 \delta^* I_2) \land (I_3 \delta^* I_4) \).

If an instruction exists which has no dependencies within the basic block, that instruction may be placed lexically anywhere 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 \( \exists I_i = \text{branch} \land I_i \notin (I_i \delta I_i) \).

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.

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.

<table>
<thead>
<tr>
<th>Part A</th>
<th>Part B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$ LOAD [R2+R4],R1</td>
<td>JMP 105</td>
</tr>
<tr>
<td>$I_2$ ADD R4,R5,R3</td>
<td>LOAD [R2+R4],R1</td>
</tr>
<tr>
<td>$I_3$ JMP 105</td>
<td>ADD R4,R5,R3</td>
</tr>
<tr>
<td>$I_4$ STORE R3,[R5+R7]</td>
<td>STORE R3,[R5+R7]</td>
</tr>
<tr>
<td>$I_5$ SUB R5,R2,R9</td>
<td>SUB R5,R2,R9</td>
</tr>
<tr>
<td>$I_6$ MOV R2,R4</td>
<td>MOV R2,R4</td>
</tr>
</tbody>
</table>

Figure 3.6: Delay branching

Notice, in part A of figure 3.6, that the execution of $I_3$, the JMP instruction,
will demand that the next instruction to execute is $I_{105}$. At the same
time, instructions $I_4$, $I_5$ and $I_6$ are being processed by the pipeline. Since
the next instruction to execute after the jump will be $I_{105}$ the pipeline would
halt, $I_4$, $I_5$ and $I_6$ would be flushed, and $I_{105}$ 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 $I_1$. When
$I_1$, the JMP, is executed it will demand that the next instruction to enter
the pipeline is $I_{105}$ 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

\[\text{begin}\]
\[\cdot \text{compute direction and distance vectors}\]
\[\cdot \text{create ISDG.}\]
\[\text{if } (\text{direction vectors} = \text{undefined}) \text{ then}\]
\[\cdot \text{do nothing;}\]
\[\text{else}\]
\[\text{if } (\text{loops are tightly nested}) \text{ and}\]
\[\quad (\text{loop limits of inner and outer loops are invariant}) \text{ and}\]
\[\quad (\text{there is no dependence relation} S_i \delta_{<,>}^* S_j) \text{ then}\]
\[\quad \cdot \text{perform loop interchange}\]
\[\text{if } (\text{loop is a single loop or the most inner nested loop}) \text{ and}\]
\[\quad (\text{cycles exist in ISDG}) \text{ and}\]
\[\quad (\text{there is no dependence relation} S_i \delta_{(=)}^* S_j) \text{ then}\]
\[\quad \cdot \text{perform loop code motion;}\]
\[\text{end}\]
2: Basic Block Transformation

For all basic blocks do

\[ \text{begin} \]

- 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 \( \text{WINDOW} = \) first node and let the \( \text{WINDOW} \) size be \( N \). Where \( N \) is the number of active instructions allowed in the pipeline at anyone time.

\[ \text{if} \ \text{that node does not exist then} \]

- goto 3;
- Give each node in the PDG a weight = to the number of children it has.
- Initially, let \( \text{CANDIDATE-SET} \) represent the instructions that have no incoming arcs, do not include the delay branch instruction.

\[ \text{while} \ \text{WINDOW} \neq \text{empty} \ \text{do} \]

\[ \text{begin} \]

\[ \text{while} \ \text{WINDOW} \ \text{has dependence} \ \text{do} \]

\[ \text{begin} \]

50
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

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 \(D_B\).

\[
if (D_B = NOP) \text{ then} \\
\quad \bullet \text{ replace the } D_B \text{ with the delay branch instruction.}
\]

\[
else \\
\quad \bullet \text{ Insert delay branch instruction after } D_B.
\]

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 \(P_{4,4}\) where all hazards occur in adjacent slots of the pipeline. That is between \(I_1\) and \(I_2\) then \(I_2\) and \(I_3\) \(\ldots\) \(I_{n-1}\) 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

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

PDG

\[
\begin{array}{c}
\text{IF} \quad \text{ID} \quad \text{OD} \quad \text{EX} \\
\begin{array}{cccc}
t_0 & I_1 & I_2 & I_1 \\
t_1 & I_2 & I_2 & I_1 \\
t_2 & I_3 & I_2 & I_1 \\
t_3 & I_4 & I_3 & I_2 \leftarrow I_1 \\
t_4 & I_5 & I_4 & I_3 & I_2 \\
t_5 & I_5 & I_4 & I_3 & I_2 \\
t_6 & I_5 & I_4 & I_3 & I_2 \\
t_7 & I_5 & I_4 & I_3 & I_2 \\
t_8 & I_6 & I_5 & I_4 \leftarrow I_3 \\
t_9 & I_7 & I_6 & I_5 & I_4 \\
t_{10} & I_8 & I_7 & I_6 \leftarrow I_5 \\
t_{11} & I_9 & I_8 & I_7 & I_6 \\
t_{12} & I_{10} & I_9 & I_8 & I_7 \\
t_{13} & I_{11} & I_{10} & I_9 & I_8
\end{array}
\end{array}
\]
Figure 3.9: Example code before transformation.

Figure 3.10: Example Code after basic block transformation.
Figure 3.11: Example code after instruction timing transformation.
Figure 3.12: Example code after branch delay transformation.

The rescheduled code shown in figure 3.12 is now ready for that specific pipeline. All possible 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 exists if \( \exists I_i \text{ var := con} \wedge \exists I_j \in (I_i \delta I_j) \).

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 \( I_1 R_3 := \text{var} \wedge (I_1 \delta I_2) \). When folding is applied, as shown in part B of figure 3.13, the dependence is broken and parallelism is created between \( I_1 \) and \( I_2 \). The transformation substituted the value of \( R_3 \) from instruction 1, for \( R_1 \) in instruction 2.

3.7.2 Dead code elimination

Definition 3.1 Dead code exits if

\[ \exists I_j \text{ such that } (I_j \in (I_i \delta I_j) \lor (I_j = I_1)) \wedge \not\exists I_k \in (I_j \delta I_k) \]
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 instruction 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 $(I_j = I_1) \land \not\exists I_k \in (I_1 \delta I_k)$.

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 affects the overall performance of the pipeline.
Bibliography


