Code Scheduling Methods for Some Architectural Features in PIPE

by

James R. Goodman
Honesty C. Young

Computer Sciences Technical Report #579

February 1985

CODE SCHEDULING METHODS FOR SOME ARCHITECTURAL FEATURES IN PIPE

James R. Goodman and Honesty C. Young
Member, IEEE Student Member, IEEE

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract- **PIPE [SPKG83] [CGKP83] [GHLP85] is a very-high-performance computer architecture intended for heavily pipelined VLSI implementation. A number of architectural queues are included in the PIPE architecture to reduce the influence of delay due to accessing memory. The *prepare-to-branch* instruction is a mechanism to decrease the penalty incurred by conditional branches. We have developed a compiler for a subset of Pascal. A code scheduler is used to reorder the compiled code to take advantage of the architectural queues and the prepare-to-branch instruction. For well-structured code (loops), *software pipelining* is a technique that can take advantage further of the aforementioned features. We present the scheduling methods for automatic code scheduling and software pipelining, which can be applied to most register-register pipelined architectures by simply changing the cost table. We show the functions of these scheduling methods by examples and we illustrate some simulation results.**

Key Words and Phrases- **Code Optimization, Compiler, Computer Architecture, Code Scheduling, Dependency Graph, Instruction Reordering, Pipelined Processor, Software Pipelining, VLSI.**

1. Introduction

Many studies [Amda67] [Lee80] [Buch83] [Worl84] have shown that even in highly vectorizable programs, a slow mode (*i.e.*, the scalar mode) dominates total execution time. Thus, it is crucial that all modern vector processors be augmented with a powerful scalar processor. Pipelining has been used to build processors with a fast scalar mode for quite some time [Kogg81]. The performance of pipelined machines, however, is often limited by memory accessing and conditional branching. A new architecture, called PIPE (Parallel Instructions and Pipelined Execution) [SPKG83] [CGKP83] [GHLP85], is a VLSI-oriented, high performance architecture project at the University of Wisconsin. The primary goal of the PIPE architecture is fast execution of sequential programs. Architectural data queues are included in the PIPE architecture to reduce the influence of delay due to accessing memory. The *prepare-to-branch* (PBR) instruction is a mechanism to decrease the penalty incurred by conditional branches.

In order to achieve a high instruction issue rate, the major design principle of PIPE is simple issue conditions. That is, only a few simple conditions must be checked to initiate an instruction. The instruction itself (*e.g.*, floating point multiplication), however, may take several clock periods to complete.

A compiler for a subset of Pascal has been developed to evaluate aspects of the PIPE architecture. A code scheduler is used to reshape the compiled code to take advantage of the data queues and the PBR instruction. We have run a set of eight benchmark programs to study the impact of the aforementioned features on system performance [YoGo84a]. *Software pipelining* [Char81] [YoGo84b] (also known as *loop folding* [Weis84]) is a technique to schedule the code sequence of the inner-most loops by rearranging the code across basic block[1] boundaries. In this paper, we describe the methods used for code scheduling and software pipelining. The methods described in this paper are applicable to most register-register pipelined architectures by simply changing the cost table.

The PIPE architecture and the rationales behind the architectural queues and the prepare-to-branch instruction are described in section 2. The basic structure of the compiler is outlined in section 3. The code scheduler methods for the intermediate language (IL) and the machine code are discussed in section 4 and 5, respectively. The concept of software pipelining is explored in section 6. Some simulation results are presented in section 7. Related research and conclusions are presented in section 8.

2. The PIPE Architecture

We briefly describe the relevant details of the PIPE architecture to make this paper self-contained. A more complete description is available elsewhere [SPKG83] [CGKP83] [GHLP85].

(1) Most instructions are of a 3-address form. Only LOAD's and STORE's are used to access memory.

[1] A basic block is a code sequence with no jumps in except at the beginning and no jumps out except at the end [AhUl77]

(2) Several execution modes are possible depending on the organization and the number of PIPE processors in the system. For this discussion, we will consider primarily the *single processor* (SP) mode where there is only one processor in the system. The instruction set for PIPE in SP mode is comparable in style to the CDC-6600 [Thor70], with the addition of the queues and the PBR instruction. A block diagram of a single PIPE processor is shown in Fig. 1.

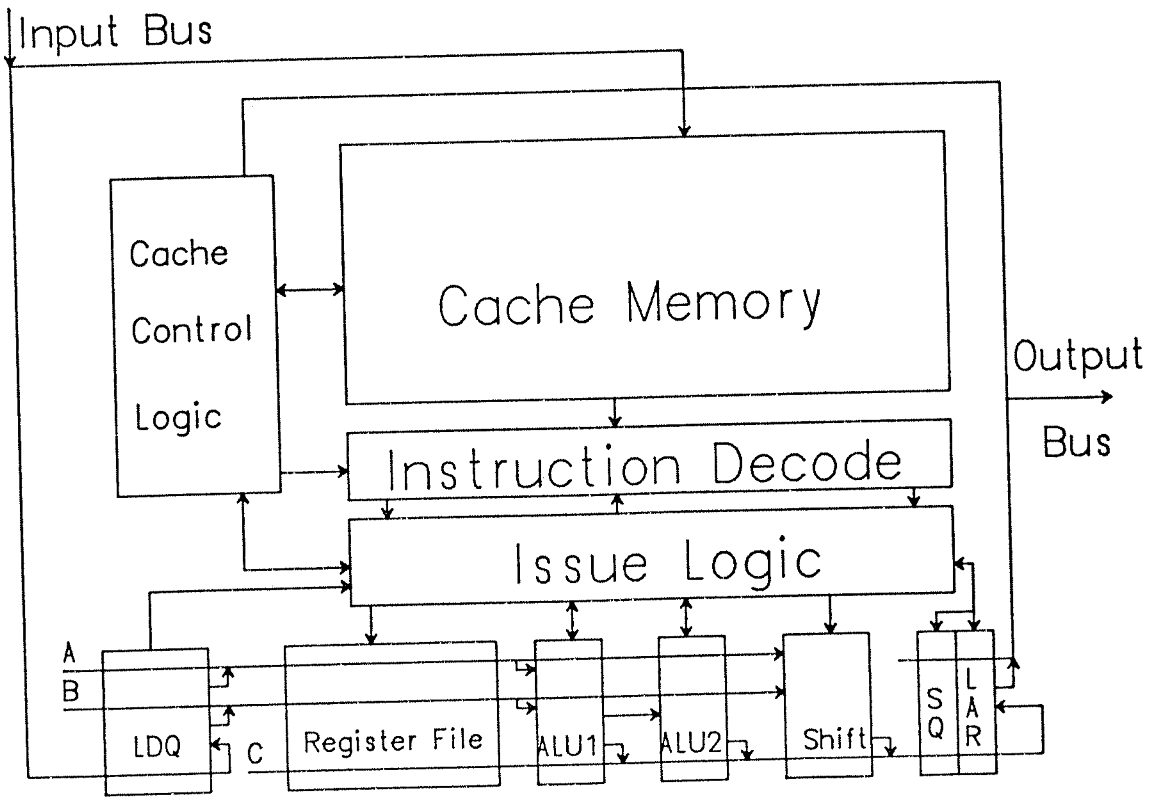

Fig. 1. A PIPE Processor

(3) The interface between the processors and memory subsystem is a set of architectural queues. When a LOAD instruction is executed, the data specified will be returned via the *Load Data Queue* (LDQ) as its last element. An additional instruction is needed to move an operand to one of the general purpose registers if the compiler decides to keep that operand in a register. Because of the three address instruction format, either one or two elements can be removed from the queue at a time. To simplify the implementation, we restrict generated code to make only one reference to the load queue per instruction [CGKP83]. A STORE is accomplished by the following two primitive instructions: (1) the store address is put on the *Store Address Queue* (SAQ); (2), the data is put on the *Store Data Queue* (SDQ). The order of these two instructions is irrelevant, and other instructions may occur in between, including multiple instances of the first instruction. There are no queue manipulation instructions. At the architectural level, the heads and tails of these queues appear as registers. A specific register (*R7* is used for the discussion of this section) in a register field designates one of the queues, instead of register 7. (That is, general purpose register *R7* is not accessible to the programmer.) Thus, the queues are implied by the instruction. For example, *R7* as the source register means LDQ while *R7* as the destination register means SDQ. SAQ is implied by all STORE instructions, and LAQ by all LOAD instructions. The rationales of these queues are explicated in section 2.1.

(4) *Prepare-to-Branch* (PBR) is used in PIPE. The meaning of the PBR instruction is explained in section 2.2.

2.1. The Load Data Queue

Among all the queues, the Load Data Queue (LDQ) is of particular interest. In general, the execution time of a memory reference instruction (LOAD/STORE) is longer than that of a register-register instruction. In particular, the delay due to a load in a pipelined computer has great influence on the performance of the entire system. The LDQ is a mechanism to separate the request of an operand (*i.e.*, the LOAD instruction) from its use, thus reducing the urgency of memory read operations. Throughout this paper, the LOAD instruction is used to represent memory accessing operations and LDQ is used to indicate relevant queues, whenever appropriate. The advantages of the LDQ are:

(1) The LDQ serves as a window between the CPU and the memory system. One can view the LDQ as a set of *queue registers* in addition to the general purpose registers.

(2) There is no need to allocate registers for *use-once operands*. The use-once operands are taken out of the LDQ directly. In many programming environments, most operands are used either once or repeatedly. A smart compiler should allocate registers only for latter type of operands. An additional move instruction is needed for operands which are allocated in registers.

(3) Data can be requested several cycles before it is needed. The code scheduler can reorder the code to overlap the data transmission and access time with the execution of other instructions. This overlap is achieved by moving instructions between the LOAD instruction and the instruction that uses the operand. This is particularly important for tight loops.

(4) It is more flexible for the code scheduler to schedule the code with the presence of the LDQ than the case where the scheduling is limited to a set of general purpose registers. There is no data dependence, at the register level, between use-once operands (except that they must be fetched in the order they are used). The same variable for different iterations can be fetched without register conflicts. The software pipelining mechanism described in this paper is an automatic way of prefetching operands across basic block boundaries.

(5) The slowness and irregularity of memory accesses can be hidden, and bursts of memory requests can be smoothed out.

The major disadvantage of the LDQ is that processor deadlock is possible for a queue of finite length. The code scheduler must guarantee a deadlock-free code sequence. Also, twice-used variables are expensive.

2.2. Prepare-To-Branch

In general, a branch instruction can be subdivided into three parts:

(1) Calculate the branch target address.

(2) Determine the branch outcome.

(3) Transfer control (if branch is taken).

One of the major goals in designing a CPU pipeline is to ensure a steady flow of instructions to the issue logic [LeSm84]. It is well known [AnST67] [Flyn72] [RiFo72] [Smit81] [LeSm84] that in a highly parallel computer system, branch instructions generally break the smooth flow of instruction fetching and execution. This results in delay, because a successful branch (a branch that is taken) changes the location of instruction fetches and because the issuing of instructions must often wait until conditional branch decisions are made. Even an unsuccessful branch (a branch not taken) or an unconditional branch usually interrupts the smooth flow of instructions.

PIPE was defined to allow the separation of these three parts. The first is actually a separate instruction so that it can be moved outside a loop. The branch target address is stored in a branch register via a move instruction. (In PIPE, in addition to the general purpose registers, there is a set of branch registers. The branch registers are used to hold the branch targets.) The last two are a single instruction, but the effects are separated in time. The PBR instruction decides the branch outcome and specifies a delay (in terms of number of subsequent instruction parcels[2] to be issued) before transfer of control. The number of instruction parcels that should be executed unconditionally after a PBR is specified by a field of the PBR instruction (called the *branch count* (BC) field). The transfer of control occurs after BC instruction parcels following the PBR have been issued. We insert a pseudo-instruction, XBR (XBR denotes the eXit point of the BRanch instruction), at the assembly language level to indicate the exit point. That is, the transfer of control for a successful branch happens at the place where XBR stands. The prepare-to-branch instruction is simply a gen-

[2] An instruction parcel is a 16-bit quantity.

eralized form of the delayed branch used in many machine organizations (*e.g.*, 801 [Radi82], MIPS [HJBG81], RISC [PaSe81] [PaSe82]).

The advantages of the PBR instruction:

(1) It is a mechanism to separate the branch decision from the transfer of control. Thus, it is possible to do *guaranteed pre-decoding* in the sense that the pre-decoded instruction is always executed. In the case of an instruction cache miss, the cache controller can do *guaranteed (cache) prefetching*.

(2) There is no penalty on the program size in using this generalized delayed branch. That is, it is never necessary to insert NOP's after the PBR instruction. For some architectures with a delayed branch (*e.g.*, RISC-I [PaSe81]), the number of instructions must be executed after a delayed jump is fixed. Under certain circumstances, NOP's have to be inserted after the delayed jump instructions.

(3) It is not necessary to fetch instructions that won't be executed.

(4) Most branch targets, stored in the branch registers, are loop invariants. Thus, standard optimization techniques [AhUl77] can be used to move the calculation of branch targets and assignment of (loop-invariant) branch registers out of the loops.

3. The PIPE Pascal Compiler

Currently, there is one high level language compiler for the PIPE architecture. The source language is a subset of Pascal. The compiler has five phases. Phase one generates intermediate language (IL) in five-tuple form, which is the quadruple form [AhUl77] augmented with the fifth attribute-tuple. The attributes are used to keep track of the original high level language structures. Phase two performs data flow optimizations [AhUl77] on the IL. Phase three schedules the IL. Phase four generates the machine code. Phase five does the machine code level scheduling to take advantage of the architectural queues and the prepare-to-branch instruction.

We do two-level scheduling for the following reasons:

(1) There are less structural dependencies at the IL level than those at the machine code level. Since we assume an infinite number of pseudo registers at the IL level, all pseudo registers are assigned at most once. (We shall see in section 6 that the software pipelining method duplicates some of the code within the body of a loop. If we do software pipelining at the IL level, a given pseudo register may be systematically assigned more than once in different basic blocks.) Thus, read-after-write (RAW) and write-after-read (WAR) hazards due to register allocation are deferred until register binding time. The register allocation algorithm can take the variable hazards into account and allocate registers accordingly.

(2) The time complexities are $O(n\log n)$[3] and $O(n)$ for scheduling methods at the IL level and the machine code level, respectively. The time complexity of one-level machine code scheduling is $O(n^2)$ (see section 5). Under the two-level scheduling scheme, the sequence of operand requests is reordered at the IL level. Thus, there is no need to change the order of memory-accessing instructions at the machine code level.

Although the IL was designed with the PIPE architecture in mind, the novel features of PIPE are not expressed explicitly. The code generated by the front-end, however, must meet the following criteria to effectively schedule the IL.

(1) The branch condition has to be in a pseudo register. That is, the branch condition is computed before the branch instruction itself. Thus, it is possible to separate the evaluation of the branch condition from the branch instruction.

(2) The index to access an array element has to be in a pseudo register. Thus, it is possible to separate the index calculation from the array element accessing.

(3) Operands for arithmetic/logic operations have to be in pseudo registers. The corresponding load (MOVE from memory) instructions are attributed with a special flag which intimates the

[3] Base 2 logarithm is used through out this paper.

code generator that these operands are used directly from the LDQ.

A complex (high level language) expression is subdivided into multiple simple IL instructions, each of them is either a binary or a unary operation. Thus, independent complex expressions may be scheduled to execute in an interleaved fashion. The code scheduler at the IL level makes a simple assumption about real variables (as opposed to pseudo registers) in that one object is not referenced by two or more different names. We also assume that an array, not each element of an array, is an object. Thus, we treat $A[i]$ and $A[j]$ as the same object regardless of the relationship between i and j. That is, we ignore the aliasing problem for the purpose of this study. Some of the restrictions mentioned above can be lifted by using sophisticated algorithms, such as the disambiguator described by Fisher *et. al.* [FERN84].

4. Code Scheduler for the Intermediate Language

The code scheduling method used at the IL level is explicated in this section. This scheduling method is similar to that at the machine level. We show the function of the scheduler by an example.

4.1. The Algorithm

Definition 1. A *weighted dependency graph* (WDG) G = $\{V,E,T,C\}$ is a weight directed acyclic graph where (a) V is a set of vertices corresponding to instructions, (b) E is a set of edges expressing data/control dependencies between pairs of vertices, (c) T is a function mapping from E to a set of non-negative integers, and (d) C is another function mapping from V to a set of non-negative integers.
□

We will use the terms "vertex" and "the instruction represented by that vertex" interchangeably, when the meaning is clear from the context. The edges are the different dependencies (to be defined later) between vertices. An edge e_{ijk} is the k-th dependency from vertex v_i to v_j. If there is an edge e_{ijk} pointing from vertex v_i to vertex v_j the value t_{ijk} associated with the edge is the time[4] from the issuing of v_i till the k-th dependency of v_i has been lifted. Note that it is possible to have more than one edge between a pair of vertices. Each edge represents a different dependency and the costs with these edges may vary. All dependency edges are pointing forward in the sense that a given

[4] This is an estimated time. It is very difficult, if not impossible, to get the actual execution time of each instruction because of some asynchronous operations (*e.g.*, memory bank conflicts, cache misses)

instruction is dependent only on instructions that appear earlier in the textual order. The cumulative cost c_i associated with the vertex v_i is the minimum amount of time from the issuing of v_i to the end of the WDG.

Definition 2. Given two vertices (instructions) v_i and v_j in a weighted dependency graph, and v_i precedes v_j in the original sequence.
(1) V_j is *RAR* (read-after-read) dependent on v_i iff both instructions get values from the same object.
(2) V_j is *RAW* (read-after-write) dependent on v_i iff v_i assigns a value to an object and v_j gets a value from the same object.
(3) V_j is *WAR* (write-after-read) dependent on v_i iff v_j assigns a value to an object and v_i gets a value from the same object.
(4) V_j is *WAW* (write-after-write) dependent on v_i iff both instructions assign values to the same object.
(5) V_j is *CON* (control) dependent on v_i iff v_i must precede v_j logically.
□

The objects at the IL level, such as variables in a program and pseudo registers, are independent of the organization of the processor. The objects at the machine code level, however, are processor resources, such as registers, queues and the memory system. It is relatively difficult to distinguish an arbitrary memory element from another at the machine code level. Hence, the memory system, rather than the elements of the memory system, is considered a single object. This constraint will not affect the code quality reordered by the machine level scheduler because the IL scheduler has put all memory references in the proper order.

Because a basic block, at the IL level, ends with a branch, this branch has to be the last instruction even after the scheduling. Thus, there are control dependencies between all other instructions in that basic block and this branch. In other words, a control transfer point induces control dependency between instructions before and after it. As mentioned before, we introduce a pseudo instruction XBR as a place holder for the exit point of the prepare-to-branch instruction. At the machine language level, all other instructions within the same basic block are *CON* dependent on the XBR pseudo instruction (not the PBR instruction). Thus, no instructions will be moved to follow the XBR instruction. Instructions, however, may be moved to follow the PBR. In fact, one of the purposes of the machine code scheduler is to move suitable instructions to follow the PBR instruction.

RAR and WAW dependencies are for the machine code scheduling method only, for the reasons explained below. In general, an object can be read many times and still hold the same value. Hence, the order of reading from that object is irrelevant. Therefore, RAR does not exist between objects at the IL level. There is at least one exception in that if the object is a queue, the order of reading is important. Suppose both v_i and v_j read from the same queue and v_i precedes v_j. Then v_i gets the first element of that queue where v_j gets the second one. The effect is different if we swap the order of v_i and v_j. The same argument holds for WAW hazard on writing to a queue. If an object at the IL level is assigned twice without being used between the two assignments, the first assignment is useless, hence can be removed (by the compiler). That is, the WAW hazard does not exist at the IL level, either. Consequently, we don't have to check for RAR and WAW dependencies for scheduling done at the IL level.

It has been shown that the reorganization problem with strict limitations is *NP-complete* [HeGr83]. Thus, heuristics are used in developing algorithms in this study.

```
Algorithm 1. Intermediate Language Code Scheduling.
    foreach basic block do
        (1-1) build data/control dependency graph;
        (1-2) assign cost to each instruction in reversed topological sort order;
        (1-3) issue instructions according to weighted topological sort order;
    od;
                                                                              □
Algorithm 2. Build Data/Control Dependency Graph.
    foreach instruction in the original textual order do
        (2-1) add a vertex corresponding to this instruction to the dependency graph;
        (2-2) forall objects the current instruction defines (writes to) do
            (2-2-1) add a WAR edge from the last previous instruction that uses (reads) the
            object to the current instruction;
            (2-2-2) assign a cost to the edge;
        od;
        (2-3) forall objects the current instruction uses (reads from) do
            (2-3-1) add a RAW edge from the last previous instruction that defines (writes) the
            object to the current instruction;
            (2-3-2) assign a cost to the edge;
        od;
        (2-4) if the last instruction of the current basic block is a branch instruction then
            (2-4-1) add a CON edge from the current instruction to the branch instruction;
        fi;
    od;
                                                                              □
```

Lemma 1. There is a one-to-one correspondence between input instructions and vertices of the dependency graph.

Proof. From statement (2-1).

□

Lemma 2. All dependencies, at the IL level, are included in the dependency graph constructed by Algorithm 2.

Proof. From Definition 2 and statement (2-2), (2-3), and (2-4).

□

For the discussion of this section, we assume the current basic block size is n. That is, there are n instructions in the current basic block.

Lemma 3. The time complexity of Algorithm 2 is linear in the size of input.

Proof. Since an instruction defines only a (small) finite number of objects, the maximal number of objects an instruction can define is dependent on the design of the instruction set but independent of the input program size. Let the maximum numbers of objects that can be defined and used by an instruction be D and U, respectively. Thus, statement (2-2) is executed at most $D \cdot U$ times per iteration. Similarly, statement (2-3) is executed at most $D \cdot U$ times per iteration. Statement (2-4) is executed once per iteration. A scheme similar to hashing can be used to find the last previous define (use) of an object in constant time. Let C_s be the time required to find the last previous define (use). Therefore, the running time of Algorithm 2 is bounded by $(2 \cdot D \cdot U \cdot C_s + C_c) \cdot n$, where C_c is the time to add a control dependency. In other words, the time complexity of Algorithm 1 is $O(n)$:

□

For practical purposes, the values of D and U are small. In a typical three-address form, the values for D and U are 1 and 2, respectively. Thus, $D \cdot U$ is still a very small number.

```
Algorithm 3  Assign Cost to a Dependency Graph.
{ The cumulative cost of an instruction $v_i$ is designated by $C(v_i)$. }
    (3-1) foreach instruction $v_i$ do
        (3-1-1) $C(v_i) <- 0$;
    od;
    (3-2) do the topological sort;
    (3-3) foreach instruction $v_i$ in the reversed topological sort order do
        (3-3-1) foreach edge $e_{ij}$ pointing out from $v_i$ to $v_j$ with cost $T(e_{ij})$ do
            (3-3-1-1) $C(v_i) <- Max2(C(v_i), C(v_j) + T(e_{ij}))$ ;
            { Max2 returns the larger one of its two arguments }
        od;
    od;
```

□

The reason we use the reversed topological sort order is that when we are computing the cumulative cost of v_i, the cumulative costs of instructions that are dependent on v_i have been evaluated earlier. In other words, in statement (3-3-1-1), $C(v_j)$ has been known and will not be changed when we are calculating $C(v_i)$.

Lemma 4. The time complexity of Algorithm 3 is linear in the size of input.

Proof. Statement (3-1) is executed n times. Except for the branch instructions, there are at most $D \cdot U$ edges pointing to any vertex. There are n-1 edges pointing to the branch instruction. Therefore, there are at most $((n-1) \cdot D \cdot U) + (n-1)$ edges in the WDG. The time to do the topological sort on the WDG is $O(\,|\,V\,| + |\,E\,|\,)$, hence $O(n)$ [Knut73]. The time to execute statement (3-3-1-1) is constant and statement (3-3-1-1) is executed at most $((n-1) \cdot D \cdot U) + (n-1)$ times. (Recall that $(n-1) \cdot D \cdot U) + (n-1)$ is the number of edges in the WDG.) Thus, the time complexity of (3-3) is also $O(n)$. Therefore, the time complexity of Algorithm 3 is linear in the size of input.

□

Definition 3. A *leader* of a WDG in a topological sort is a vertex with no predecessors. That is, either this vertex does not depend on any other vertices, or all its predecessors have been removed (issued).

□

Definition 4. A *weighted topological sort order* of a WDG is topological sort order subject to the following constraint. V_i and v_j are two vertices, if v_i precedes v_j, then $C(v_i) \geq C(v_j)$, where C is the cumulative cost function of a vertex.

□

Algorithm 4. Issue Instructions by Weighted Topological Sort Order.

```
(4-1) calculate the leader set;
(4-2) repeat
    (4-2-1) pick up an instruction ($v_i$) from the leader set with maximum cumulative cost;
    (4-2-2) issue $v_i$;
    (4-2-3) remove $v_i$ from the leader set;
    (4-2-4) insert new leaders to the leader set;
until the leader set is empty;
(4-3) if some instructions have not been issued then
    (4-3-1) error condition;
fi;
```

□

The error condition, (*i.e.*, statement (4-3)) should not happen at the IL level. Because the original input is a legal sequence.

Lemma 5. The time complexity of Algorithm 4 is $O(n\log n)$.

Proof. The data structure used for the leader-set is an AVL tree [Knut75], which has $O(\log m)$ time complexity to insert, delete, and find the maximal element, where m is the size of the AVL tree. The size of the leader set is no bigger than n and each vertex (instruction) is put on the leader set only once. Similarly, a vertex is removed from the set only once. Thus, each of the statements (4-2-1), (4-2-3) and (4-2-4) is executed at most n times. The original leader set can be built with time complexity $O(n\log n)$. Therefore, the time complexity of Algorithm 4 is $O(n\log n)$.

□

Lemma 6. If Algorithm 4 terminates successfully, there is a one-to-one corresponds between vertices of the WDG and the output instructions. In other words, the output instructions are a permutation of vertices of the WDG.

Proof. If Algorithm 4 terminates successfully, then all vertices have been issued. On the other hand, instructions issued by Algorithm 4 are members of the leader set which is constructed from vertices of the WDG.

□

Lemma 7. All dependencies represented by the WDG are preserved in the output instruction stream.

Proof. From the properties of topological sort.

□

Theorem 1. The time complexity of Algorithm 1 is $O(n\log n)$.

Proof. From Lemma 2, 3, and 4, the time complexity for one iteration of the loop in Algorithm 1 is $O(n\log n)$. Let n_i be the size of the i-th basic block and N be the size of input program. That is

$$N = \sum_i n_i$$

Thus, the time complexity of Algorithm 1 is

$$O(\sum_i n_i \log n_i) \le O(\sum_i n_i \log N) = O(N\log N)$$

□

Definition 5. For two code sequences, C_i and C_j, C_i is said to be logically equivalent to C_j iff C_i is a permutation of C_j and all dependencies in C_i are preserved in C_j.

□

Theorem 2. The scheduled code (the output of Algorithm 1) C_{sch} is logically equivalent to the original code (the input to Algorithm 1) C_{in}.

Proof.
If-part:
[C_{sch} is a permutation of C_{in}] From Lemma 1 and Lemma 6.
[All dependencies in C_{in} are preserved in C_{sch}]: From Lemma 2 and Lemma 7.
Only-if-part: Similar to that of the if-part by introducing comparable lemmas.

□

4.2. Examples

We assume the following timing for the relevant functions for the discussion of examples in this paper.

Function	Time (clock periods)
Load from Memory	6
Store to Memory	1
Branch (taken)	6
Integer Add	1
Floating Point Add	4
Floating Point Multiplication	4
Load Branch Register	1

It may take more than one clock period for an operand to be written to the memory system. As far as the issue logic is concerned, however, a store operation does not cause any subsequent instructions being blocked owing to data dependency. The time for a branch is the number of clock periods required from the issue time of the branch instruction to the issue time of the first instruction of the branch target for a successful branch. We also assume the issue logic is capable of issuing one instruction every clock period providing there are no dependencies between instructions.

Example 1. We will use the following example [HwBr84] throughout this paper.

```
          DO 999 I = 1, 1000
    999        Y(I) = F(I) × Y(I-1) + G(I)
```

This loop is hard to vectorize on some supercomputers because of the first order linear recurrence. Some notations used in the IL are briefed below: (a) %R*n* stands for the *n*-th pseudo register at the IL level; (b) *x*[*y*] denotes an access to array *x* with offset *y*; and (c) An "M" in the attribute tuple of a memory-accessing MOVE instruction gives a hint to the code generator that such operand will be used directly from the LDQ. The compiled IL is:

```
            MOV     %R6.1000..           /* loop bound
            MOV     %R3.y-1[%R1]..M      /* Y(0)
    LOOP
            MOV     %R2.f[%R1]..M        /* F(I)
            MULF    %R4.%R2.%R3.         /* F(I)×Y(I-1)
            MOV     %R5.g[%R1]..M        /* G(I)
            ADDF    %R3.%R4.%R5.         /* ...+G(I)
            MOV     y[%R1].%R3..         /* store the result
            ADDI    %R1.%R1.1.           /* increment loop count
            BRLE    %R1.%R6.LOOP.        /* check loop bound
```

The code generated from the above IL is:

```
               ....
               R0      ←   1000          /* loop bound
               BR0     ←   LOOP          /* branch target
               LDQ     ←   R1, y-1       /* load Y(0)
               R3      ←   LDQ           /* move Y(0) to R3
    LOOP
          S1   LDQ     ←   R1, f         /* load F(I)
          S2   R2      ←   LDQ ×f R3     /* F(I)×Y(I)
          S3   LDQ     ←   R1, g         /* load G(I)
          S4   SAQ     ←   R1, y         /* addr of Y(I)
          S5   R3      ←   R2 +f LDQ     /* ...+G(I)
          S6   SDQ     ←   R3            /* value for Y(I)
          S7   R1      ←   R1 + 1        /* increment loop count
          S8   R4      ←   R1 − R0       /* test bound
          S9   PBRLE       R4, BR0, 0    /* branch back
          S10  XBR
```

The issue time and the completion time of the above code sequence are:

Statement Number	Issue Time	Completion Time
S1	0	6
S2	6	10
S3	7	13
S4	8	9
S5	13	17
S6	17	18
S7	18	19
S8	19	20
S9	20	**26**

As a convention used in this paper, the boldfaced number in the completion-time column indicates the effective execution time (in terms of clock periods) per iteration. The scheduled code from Algorithm 1 is:

```
             ....
LOOP
             MOV      %R2,f[%R1],,M
             MOV      %R5,g[%R1],,M
             MULF     %R4,%R2,%R3,
             ADDF     %R3,%R4,%R5,
             MOV      y[%R1],%R3,,
             ADDI     %R1,%R1,1,
             BRLE     %R1,%R6,LOOP,
```

The machine code generated from the scheduled IL code is:

```
            ....
LOOP
            S1     LDQ       ←    R1, f
            S3     LDQ       ←    R1, g
            S2     R2        ←    LDQ ×f R3
            S4     SAQ       ←    R1, y
            S5     R3        ←    R2 +f LDQ
            S6     SDQ       ←    R3
            S7     R1        ←    R1 + 1
            S8     R4        ←    R1 − R0
            S9     PBRLE          R4, BR0, 0
            S10    XBR
```

The issue time and the completion time are:

Statement Number	Issue Time	Completion Time
S1	0	6
S2	1	7
S3	6	10
S4	7	8
S5	10	14
S6	14	15
S7	15	16
S8	16	17
S9	17	23

□

We didn't intend to include many architectural features within the IL. In particular, the concepts of prepare-to-branch and data queues are not delineated. Thus, the machine level code shown above is not optimal. Especially, the prepare-to-branch instruction is not utilized at all. We shall see in the next section that the code scheduler at the machine code level takes advantage of the special architectural features.

5. Code Scheduler at the Machine Code Level

The algorithm used for the machine code level scheduler is similar to that used in the IL level scheduler. The major differences are (a) all objects are processor resources, such as registers; (b) RAR and WAW hazards are used to enforced the first-in-first-out (FIFO) order of queues for the reasons explained earlier; and (c) an additional restriction (described below) is imposed for a queue-filling instruction (instruction that puts an element on a queue) to become a leader, viz., this queue-filling instruction should not overflow the queue. The last difference guarantees that the scheduled code is deadlock-free. It is, however, possible to have an error condition at the machine code level because of different assumptions about queue sizes. The functions of the machine code scheduler are: (1) to utilize the architectural queues; (2) to utilize the prepare-to-branch instruction; (3) to enforce the FIFO nature of the queues; and (4) to guarantee deadlock free code for queues of finite size. From the different constraints at the machine code level, we have the following theorem.

Lemma 8. The time complexity of Algorithm 4 at the machine level is linear in the size of input.

Proof. The number of elements in a leader-set, for the scheduling method at the machine code level, is dependent on the number of objects (thus processor resources) in the system but independent of the input program size. Thus, the size of the leader-set in Algorithm 4 is bounded by a constant which depends only on the processor organization. Therefore, it takes constant time to insert an element, delete an element, and find the maximal element from the leader set. We have to do the insert/delete/find operations at most *n* time in Algorithm 4. Hence, the time complexity of Algorithm 4 is linear in the size of input program.

□

Theorem 3. The time complexity of scheduling method used at the machine language level is linear in the size of the input program.

Proof. From Lemma 3, 4, 8, and superposition.

□

Example 2. The scheduled code of the machine code in Example 1 is

```
             ....
LOOP
             S1    LDQ      -    R1, f
             S2    SAQ      -    R1, y
             S3    LDQ      -    R1, g
             S4    R1       -    R1 + 1
             S5    R4       -    R1 - R0
             S6    R2       -    LDQ ×f R3
             S7    PBRLE         R4, BR0, 2
             S8    R3       -    R2 +f LDQ
             S9    SDQ      -    R3
             S10   XBR
```

The issue time and completion time of the code sequence are

Statement Number	Issue Time	Completion Time
S1	0	6
S2	1	2
S3	2	8
S4	3	4
S5	4	5
S6	6	10
S7	7	**13**
S8	10	14
S9	11	12

S7 completes at time 13 (shown in boldface in the table), which is the number of clock periods to execute an iteration of the loop. Although S8 completes at time 14, the result generated by S8 will not be used until time 6 of the next iteration. Thus, the effective execution time is 13 clock periods per iteration.

□

Flynn [Flyn66] observed that there is always some point in the instruction fetch/decode path through which instructions pass at the maximum rate of one per clock cycle. Put differently, the maximum instruction initiating rate of an issue unit is one per clock period. This bottleneck is referred as the *Flynn limit* [GHLP85]. An examination of commercially available high performance computer systems supports this observation. The weighted topological sort method described in

Algorithm 4 does not take this limit into account. Consequently, there are cases where many independent instructions are crowded to be issued near the end of a basic block. This clustering of instructions results in unnecessarily prolonged execution time due to the Flynn limit. A simple scenario of this situation follows. Suppose there are five independent instructions with cumulative costs of two and the cumulative costs of any other instructions are larger than two. Hence, the aforementioned instructions will be the last ones to be issued according to Algorithm 4. The relative issuing order of these five instructions, however, is unimportant. As mentioned above, the cumulative cost of a given instruction is the least amount of time (in terms of clock periods) from issuing of that instruction to the end of the basic block. Thus, the best case is to finish the basic block two clock periods after issuing any of these five instructions. The underlying hardware, confined by the Flynn limit, requires 6 clock periods to complete all five instructions, which is 4 clock periods longer than we would like it to be. Some of these five instructions can be issued earlier if a slightly different scheduling method is used.

We introduce a modified scheduling method to take the Flynn limit into account. The basic idea behind the *modified weighted topological sort order* is explained here. Suppose two consecutive instructions in the weighted topological sort order with cumulative costs of n and n-k($k \geq 1$), respectively. It is possible to issue k-1 other instructions, between these two instructions, from the leader set without increasing the total execution time of the basic block. In other words, k-1 *free* time slots are available to issue instructions with smaller cumulative costs without execution time penalty. When an instruction becomes a member of the leader set, a sequence number is assigned to that instruction. This sequence number is just a form of time-stamp, which is used as a secondary key in issuing instructions. The bigger the sequence number is, the later the instruction enters the leader set. The modified topological sort order is detailed in the next Algorithm.

```
Algorithm 5. Issue Instructions by Modified Weighted Topological Sort Order.
    (5-1) calculate the leader set with proper assignment of sequence number;
    (5-2) p_cc ← 0; { previous cumulative cost }
    (5-3) c_cc ← 0; { current cumulative cost }
    (5-4) repeat
        (5-4-1) pick up an instruction ($v_i$) from the leader set with maximal cumulative cost;
```

```
            (5-4-2) p_cc ← c_cc;
            (5-4-3) c_cc ← $\bar{C}(v_i)$; { the cumulative cost of $v_i$ }
            (5-4-4) if (p_cc−c_cc) $\geq$ 2 then
                (5-4-4-1) SN_issue (p_cc−c_cc−1, $v_i$);
            else
                (5-4-4-2) issue $v_i$;
                (5-4-4-3) remove $v_i$ from the leader set;
                (5-4-4-4) insert new leaders to the leader set with proper assignment of sequence
                number;
            fi;
        until the laeder set is empty;
        (5-5) if some instructions have not been issued then
            (5-5-1) error condition;
        fi;
```
□

Algorithm 6. SN_issue (n : integer; v : vertex);

```
{ issuing at most "n" instructions, other than "v", according to the sequence number }
{ "v" must be in the leader set }
    (6-1) set tmp_leader_set to empty;
    (6-2) while (n > 0) and (|leader set| > 1) do
        (6-2-1) pick up an instruction ($v_j$) from the leader set with maximal sequence number;
        (6-2-2) if ($v_j$ $\neq$ v) then
            (6-2-2-1) issue $v_j$;
            (6-2-2-2) remove $v_j$ from the leader set;
            (6-2-2-3) c_cc ← $C(v_j)$; { cumulative cost of $v_j$ }
            (6-2-2-4) n ← n − 1;
            (6-2-2-5) insert new leaders to the tmp_leader_set with proper assignment of
            sequence number;
        fi;
    od;
    (6-3) merge tmp_leader_set to the leader set;
```
□

In Algorithm 6 when the size of the leader set is equal to one, the only element remaining in the leader set is the argument "v". The "tmp_leader_set" is used to avoid excessive unfairness. That is, instructions, which become leader in an activation of SN_issue, are not considered to be issued within the same activation of SN_issue.

Many variations of SN_issue are possible. We will not discuss alternative versions of SN_issue in this paper. The Flynn limit applies only to the hardware instruction issue unit. Thus, we have to apply this modified method to the machine code level scheduler alone. The bounded leader set assumption, used in Lemma 8, still holds. Therefore, we have the following theorem.

Theorem 4. The time complexity of the modified Algorithm 4 at the machine code level is linear in the size of input.

Proof. The proof is similar to that of Lemma 8 and Theorem 3.

□

As long as the high level language is available, the two level scheduling methods proposed is applicable. If we must schedule the assembly language program to take advantage of architectural features, a more expensive (in the sense of execution time) scheduling method must be used. The expense come from the reordering of memory accessing instructions without deadlock. This single level scheduling method is described below. For the simplicity of discussion, we will only elucidate the scheduling of load instructions and instructions that consume the operands from the LDQ. Without loss of generality, we assume that a given instruction takes either no operands or exactly m operands from the LDQ. We will use *queue-draining instructions* to refer to instructions that take operands from the LDQ. The corresponding load instructions are termed *queue-filling instructions*. We enumerate all queue-draining instructions from D_1 to D_n. Similarly, queue-filling instructions associated with D_i are numbered from F_{i1} to F_{im}. That is F_{ij} loads the j-th operand for the i-th queue-draining instruction. In order to enforce the FIFO nature of the queues without leading to deadlock, all queue-draining instructions have to be strung together by appropriate RAR dependency links. The corresponding queue-filling instructions are chained by suitable WAW dependency links. The FIFO nature is enforced by finding a total order of all queue-draining instructions. The corresponding queue-filling instructions are arranged accordingly. The goal of finding this total order is to minimize the execution time of a basic block subject to the system constraints.

Definition 6. A queue-draining instruction and its queue-filling instructions form a *FD-unit*. That is, $F_{i1}, F_{i2}, \cdots F_{im}$ and D_i form the i-th FD-unit of a basic block.

□

We state without proof the following:

(1) Since a queue-draining instruction takes m consecutive elements from the LDQ, queue-filling instructions from the same FD-unit do not intermix with queue-filling instructions from any other FD-units.

(2) The total order of queue-filling instructions of the i-th FD-unit is $F_{i1}, F_{i2}, \cdots F_{im}$. In other words, operands used by a given queue-draining instruction are always requested in the desired order.

Thus, the total order of all FD-units leads to the total order of queue-draining instructions as well as that of queue-filling instructions. The cost of the i-th FD-unit is the cumulative cost of D_i. That is, the minimum cumulative cost within that FD-unit. The following algorithm finds the total order of all FD-units.

Definition 7. Let $M_{i1}, M_{i2}, \ldots, M_{im}$, be members of a FD-unit FD_i and $M_{j1}, M_{j2}, \ldots, M_{jm}$, be members of a FD-unit FD_j. FD_j is dependent on FD_i iff there exist a x and a y, and M_{ix} is in FD_i and M_{jy} is in FD_j, such that M_{jy} transitively depends (*i.e.*, directly or indirectly depends) on M_{ix}. A *coalesced transitive closure* among FD-units is formed by the dependency relations among all FD-units.

□

```
Algorithm 7. Find the Total Order of All FD-units.
    (7-1) find the transitive closure of the WDG;
    (7-2) find the coalesced transitive closure among all FD-units;
    (7-3) build the FD-WDG according to the coalesced transitive closure; { the FD-WDG con-
    sists of all FD-units }
    (7-4) do weighted topological sort on the FD-WDG;
```

□

The output order from Algorithm 7 is a legal total order among all FD-units. If FD_i and FD_j are two adjacent FD-units according to the total order and FD_i proceeds FD_j, the following dependencies are added to the original WDG.

(1) Add a WAW link between F_{im} and F_{j1}.

(2) Add a RAR link between D_i and D_j.

This algorithm can always find a legal total order among FD-units, because the original text order is a legal one.

Theorem 5. The time complexity of Algorithm 7 is $O(n^2)$.

Proof. Since there are $O(n)$ edges for a WDG of n vertices, the algorithm of finding the transitive closure for use with *sparse relations* [HuSU77] applies. The transitive closure of the original WDG can be computed with time complexity of $O(n^2)$. The size of any given FD-unit is bounded by a small constant. Thus, the dependency between any two FD-units can be computed in constant time. There are at most $O(n)$ FD-units for a basic block of size n. Therefore, the dependency relations among all pair of FD-units (*i.e.*, the coalesced transitive closure) can be computed with time complexity of $O(n^2)$. The FD-WDG can also be built with time complexity of $O(n^2)$. The weighted topological sort can be done in time complexity $O(n \log n)$. Hence, the time complexity of Algorithm 7 is $O(n^2)$.

□

This scheduling method moves appropriate instructions, according to their cumulative costs, between the PBR and XBR subject to dependencies and system constraints. It does not, however, move as many as possible. There is no execution time penalty, however, because instructions not moved are issued on cycles where following instruction could not be issued anyway. The same argument holds for the queue-filling/draining instructions.

The branch count field of the PBR instruction should be updated in accordance with the number of instruction parcels being moved between the PBR and the XBR. This updating is done by the assembler.

One possible mode of PIPE is the access/execute (AE) mode. In AE mode, the access processor (AP) calculates all memory addresses and initiates all memory references for both processors. The execute processor (EP) does all algorithmic computations. Since the IL does not know different execution modes of PIPE, we don't have to introduce new scheduling method at the IL level for different execution modes. The scheduling method, for the AE mode, at the machine code level is similar to that for the SP mode except that the scheduler for the AE mode has to look at two instruction streams at the same time to avoid deadlock.

All code scheduling methods discussed so far are applied to one basic block at a time. There are not many instructions within a small basic block. It is obvious that code scheduling can be more effective for large basic blocks than for small ones. One example of an inherently small basic block comes from the high level language *while*-statement in that the evaluation of the Boolean expression associated with the *while*-statement forms a small basic block. However, it is possible, for a compiler, to convert a *while*-statement into an *if*-statement followed by a *repeat-until*-statement with proper adjustment to the Boolean expression, which reduces the number of small basic clocks. This example demonstrates that a compiler can do necessary transformations to make other optimization methods (in this case, code scheduling) more effective. In the next section, we will introduce a method which does code scheduling across basic block boundary.

6. Software Pipelining

Software pipelining [Char81] is the deliberate partitioning of a program loop, carried out by the compiler, into load/computation/store sequences. This allows overlapping the execution of these operations in a fashion similar to hardware pipelining. The effect of software pipelining is to increase the throughput of the hardware pipeline. This is accomplished by anticipating operand requests and overlapping their access time with computations on previously fetched operands.

Essentially, the function of code scheduling is to identify the critical path in the data dependency graph and to reorder the code sequence in order to overlap the operations of non-critical paths with the ones in the critical path. On the other hand, the function of software pipelining is to reshape the dependency graph by reconstructing the body of a loop in order to form a shorter critical path. The control dependency (*i.e.*, the branch instruction) is placed in the most appropriate location, and is often overlapped with some data dependencies.

The idea of software pipelining is explained by an example. Considering the following code segment:

```
for i := 1 to Max do
begin
    Load_i;
    Computation_i;
    Store_i;
end;
```

The space-time diagram of this loop is: (The x-axis is the time axis where the y-axis is the space axis. L, C, and S stand for Load, Computation, and Store, respectively.)

L_{i-1}			L_i			L_{i+1}		
	C_{i-1}			C_i			C_{i+1}	
		S_{i-1}			S_i			S_{i+1}

There are three "stages" in the loop shown above. There are data dependencies between $Load_i$, and $Computation_i$ as well as between $Computation_i$ and $Store_i$. Consequently, there are "bubbles" in the hardware pipeline. That is, some stages in the pipeline are not filled with useful work. An equivalent code sequence is:

```
    Load$_1$;
    Computation$_1$;
    Load$_2$;
for i := 2 to Max−1 do
    Store$_{i-1}$;
    Computation$_i$;
    Load$_{i+1}$;
od;
    Store$_{Max-1}$;
    Computation$_{Max}$;
    Store$_{Max}$;
```

For the next space-time diagram, we ignore the time to issue instructions (*i.e.*, only the execution time of instructions are shown). The space-time diagram for the new sequence looks:

L_{i-1}	L_i	L_{i+1}		
	C_{i-1}	C_i	C_{i+1}	
		S_{i-1}	S_i	S_{i+1}

There may be fewer data dependencies in the latter code sequence. Thus, the efficiency of the issue unit of the latter code sequence is higher. Consequently, the execution time of this particular loop can be reduced by software pipelining.

In PIPE, the only windows between CPU and the memory system are the architectural queues. These queues make it easier to do software pipelining because registers need not be allocated for loads and stores. Also, with queues, it is possible to load the data several iterations ahead of time (if it is beneficial to do so). This software pipelining technique can be applied to either the IL level or the machine level. Some semantic information, such as the loop boundary, may have to be kept around to simplify the job of software pipelining.

The following symbols are used for the discussion of the algorithms described in this section. Assume there are M_l instructions in the body of the loop and M of which are unrelated to loop control. That is, there are M_l-M instructions which are used for loop control. Let S_{ij} be the i-th instruction from the j-th iteration of the loop.

Definition 8. *Momentous instructions* within a loop are instructions other than the loop control ones.

□

Definition 9. The *n-th order unrolled sequence* is a sequence of momentous instructions formed by unrolling the body of a loop n times.

□

Definition 10. The *cost of a code sequence* is the time between the issuing of the first instruction in that sequence and the completion of the last one.

□

Definition 11. The *core* of a scheduled n-th order unrolled sequence is a code sequence that satisfies the following conditions.
(1) For all $1 \le i \le M$, there exists a unique j such that S_{ij} is in the sequence.
(2) For all code sequences satisfy condition (1), choose the one with minimum cost.

□

Algorithm 8. Software Pipelining.

```
(8-1) get the n-th order unrolled sequence;
(8-2) do code scheduling;
(8-3) find the core;
(8-4) reconstruct the loop;
```

□

Algorithm 9. Find the *core*.

```
foreach instruction $S_{ij}$ do
    (9-1) find a sequence, starting with $S_{ij}$, that satisfies condition (1) of a core or end-of-input-stream;
    (9-2) if end-of-input-stream then
        (9-2-1) exit;
    else
        { call this sequence $Q_{ij}$ }
        (9-2-2) calculate the cost of $Q_{ij}$;
        (9-2-3) if $Q_{ij}$ has the minimum cost up to this point then
            (9-2-3-1) record $Q_{ij}$;
        fi;
od;
{ the last recorded $Q_{ij}$ is the core found by this algorithm }
```

□

For practical purposes, the degree of unrolling (the value of n) should be at least two to make the software pipelining more effective than the simple scheduling method. It is easy to see that a core can always be found as long as the degree of unrolling is no less than one.

Lemma 9. The time complexity of Algorithm 9 is $O(n^2)$.

Proof. There are $n \cdot M$ instructions in the n-th order unrolled sequence, where n is the degree of unrolling. Normally, the degree of unrolling (*i.e.*, the value of n) is a small constant (*e.g.*, 3), and is independent of M. Thus, statement (9-1) is executed $O(M)$ times. Similarly, statement (9-2) is executed $O(M)$ times. Statement (9-2-2) and (9-2-3) can be done in constant time. Consequently, the time complexity of Algorithm 9 is $O(M^2)$.

□

Algorithm 10. Reconstruct the loop.

```
(10-1) put all instructions before the core before the body of the newly constructed loop;
(10-2) put all instructions after the core after the body of the newly constructed loop;
(10-3) adjust loop bounds;
(10-4) add loop control instructions back;
```

□

Lemma 10. The time complexity of Algorithm 10 is constant.

Proof. Trivial.

□

Theorem 6. The time complexity of Software Pipelining is $O(M^2)$, where M is the number of momentous instructions within the loop.

Proof. It takes constant time to do the following two operations: (a) get the n-th order unrolled sequence; and (b) reconstruct the loop (Lemma 10). The time complexity of code scheduling is no worse than $O(M\log M)$ (Theorem 1). From Lemma 9, the time complexity of finding the core is $O(M^2)$. Therefore, the time complexity of the Software Pipelining algorithm is $O(M^2)$.

□

The software pipelining algorithm can be applied to both the IL level and the machine code level. In either case, the time complexity is $O(M^2)$.

With minor modification, this software pipelining method can be applied to loops where the number of iterations is not known at compile time (*e.g.*, *for*-loops having variables as loop bounds; *while* or *repeat-until* loops).

Example 3. The software pipelined code of Example 1 is:

```
              ....
LOOP
              S1    SDQ      ←   R3
              S2    SAQ      ←   R1, y
              S3    R2       ←   LDQ ×f R3
              S4    R1       ←   R1 + 1
              S5    R4       ←   R1 − R0
              S6    PBRLE        R4, BR0, 5
              S7    R3       ←   R2 +f LDQ
              S8    LDQ      ←   R1, f
              S9    LDQ      ←   R1, g
              S10   XBR
```

The issue time and completion time are:

Statement Number	Issue Time	Completion Time
S1	0	1
S2	1	2
S3	2	6
S4	3	4
S5	4	5
S6	5	**11**
S7	6	10
S8	7	13
S9	8	14

Although S9 completes at time 14, the operand loaded by S9 will not be used until S7 of the next iteration. The effective speed of this loop is 11 clock periods per iteration. If the memory load delay were much longer, the software pipelining method could generate code to do operand prefetching two or more iterations ahead.

□

The efficiencies of the issue unit for the code segments shown in the previous examples are 35%, 39%, 69%, and 82%, respectively. The efficiency of the issue unit is a good indication of the throughput, hence, speed up. Thus, for this particular example, the speed-up of software pipelining over straight forward scheduling is about 19%.

It is relatively difficult to do software pipelining on loops with if-statements, because it is hard for the compiler (code scheduler) to know whether operands in the if-part should be prefetched. The compiler, however, can at least prefetch operands that are used to determined the Boolean condition of the if-statement. If the *true-ratio* of the if-statement is known, the compiler can do appropriate prefetching accordingly. This is often the case, for example, in testing for error conditions.

Software pipelining has marginal performance effects on loops where the execution time of an iteration is much longer than that of a memory reference. Thus, it is advisable to apply the software pipelining method only to loops where the execution time of an iteration is comparable with that of a memory reference. Intuitively, it is beneficial only to do software pipelining on small loops. There may not be many small loops in programs. A big loop may be split (by the compiler) into a few smaller loops to fit the loop bodies into the hardware instruction buffer. In the supercomputing environment, however, a relative big loop may be divided into vectorizable parts and non-vectorizable parts by applying high level language program transformation techniques, such as the ones in Parafrase [KKLW80]. The non-vectorizable parts of a big loop may consist of a few small loops intermixed with some vectorizable loops which may also be small. The execution time of these non-vectorizable loops tends to dominate the total execution time. Software pipelining is a good way to reduce the execution of the non-vectorizable small loop, hence, total execution time. Because the software pipelining method is invoked only for small loops, the quadratic execution time of the software pipelining algorithm will not introduce excessive overhead to the entire scheduling process.

Loop unrolling is another method to speed up loops. The major difficulties of loop unrolling are [Weis84] (a) register allocation; and (b) code size of the loop. The latter has dramatic impact on the performance of loops. When the code size of a loop exceeds the hardware instruction buffer size, the system performance degrades significantly due to excessive instruction buffer misses. The software pipelining method, on the other hand, does not affect the code size of a loop body (though the preamble may be larger). Thus, the software pipelined code does not have optimization anomalies in that the execution time of the software pipelined code is no longer than that of the original code. In the worst case, the software pipelined code is the same as the original code. Since the execution times of each instruction is used to guide the software pipelining. The degree of prefetching (*e.g.*, the number of preload operands) is flexible in the sense that different loops may have different degrees of prefetching.

7. Simulation Studies

In this section, we show some experimental results concerning the effectiveness of the aforementioned scheduling methods in utilizing the special features provided by the PIPE architecture.

7.1. Queues and PBR Instruction

A functional interpreter and a performance simulator have been built to evaluate the performance of the PIPE architecture. Our measurement method is to compare the performance of the current PIPE architecture with that of a ``bare PIPE''. By ``bare PIPE'', we mean the PIPE architecture with the degenerate case for each special feature; in particular, the LDQ size is 1, the branch count field is always 0 (*i.e.*, no instructions are moved to follow the PBR instruction).

We ran a set of eight bench programs. These benchmark programs are described below.

ACK	Ackermann function with arguments 1, 1.
FACT	Calculate the factorial by recursive calls. The argument is 5.
HEAP	A heap sort program. We picked heap sort because its run time is less data dependent than most other sorting algorithms.

LLL5[5]	The fifth loop (tri-diagonal elimination -- below diagonal) from the Livermore Loops [McMa72] [RiSc84]. We ran the loop 20 iterations instead of the original 1000 iterations.
LLL8[5]	The eighth loop (numerical solution for P.D.E.) from the Livermore Loops.
MATMUL	Matrix (of size 3 x 3) multiplication. There is no procedure call in this program.
VECADD	The sum of two vectors of size 200.
VECHAND	Hand-coded version of VECADD for the bare PIPE.

Though, it is impractical to build a very large on-chip instruction cache for the fabrication technology available to universities, we assume a large, fully-associative instruction cache with block size of four parcels. The instruction cache is much larger than any of the programs tested. Therefore, we may consider the size of the instruction cache to be infinite. Thus, when a line (cache block) is referenced for the first time, an instruction cache miss occurs. All subsequent execution of the instructions within the same line (block) will not cause any further cache misses.

Table 1. Relative performance gain by adding the LDQ and the PBR instructions to bare PIPE machine

	Bare PIPE (cycles)	Loaded PIPE (cycles)	Speed Up
ACK	1787	1356	1.32
FACT	2104	1451	1.45
HEAP	3228	2597	1.24
LLL5	5993	2893	2.07
LLL8	49443	19928	2.48
MATMUL	2067	1048	1.47
VECADD	13774	6957	1.98
VECHAND	4693	2508	1.87

[5] Since the only compiler for PIPE at the present time is a Pascal compiler, we converted these two loops to the Pascal syntax using integer arithmetic.

Table 2. Relative speed-up due to the LDQ only

	Bare PIPE	Loaded PIPE	Average Load Dist.	Speed Up
ACK	1787	1406	7.31	1.27
FACT	2104	1499	8.10	1.40
HEAP	3228	2724	3.68	1.19
LLL5	5993	2958	6.99	2.03
LLL8	49443	20468	11.11	2.41
MATMUL	2069	1514	12.39	1.36
VECADD	13774	7972	7.40	1.73
VECHAND	4693	2719	7.02	1.73

Table 3. Relative performance for different LDQ Size

	LDQ	size				
	1	2	3	4	6	∞
ACK	1742	1473	1416	1396	1372	1356
FACT	2051	1608	1527	1500	1470	1451
HEAP	3112	2649	2612	2607	2601	2597
LLL5	5808	3939	3415	2894	2893	2893
LLL8	46820	29707	24201	21661	20584	20468
MATMUL	1857	1525	1410	1408	1408	1408
VECADD	12367	8756	6957	6957	6957	6957
VECHAND	4693	2508	2508	2508	2508	2508

Table 4. Maximum number of operands in the LDQ

	Maximum Number
ACK	3
FACT	7
HEAP	4
LLL5	4
LLL8	6
MATMUL	3
VECADD	3
VECHAND	2

Table 5. Relative speed-up due to PBR only

	count=0 (cycles)	Using PBR (cycles)	Count Field Average	Speed Up
ACK	1787	1690	3.65	1.06
FACT	2104	1969	4.26	1.07
HEAP	3228	2913	2.60	1.11
LLL5	5993	5313	2.93	1.13
LLL8	49443	44917	2.84	1.10
MATMUL	2067	1836	4.72	1.13
VECADD	13774	11554	5.98	1.20
VECHAND	4693	2508	5.01	1.87

Table 6. The dynamic branch count distribution

		Branch Count (BC)
ACK	Max	7
	Mean	3.71
	Median	6
FACT	Max	7
	Mean	4.32
	Median	6
HEAP	Max	7
	Mean	2.05
	Median	0
LLL5	Max	6
	Mean	2.93
	Median	0
LLL8	Max	3
	Mean	2.84
	Median	3
MATMUL	Max	7
	Mean	4.21
	Median	5
VECADD	Max	7
	Mean	6.98
	Median	7
VECHAND	Max	7
	Mean	5.01
	Median	5

In table 1, we present the performance gain by adding the LDQ and PBR instruction to the bare PIPE. The speed-up column is the ratio of "the bare PIPE" column to "the loaded PIPE" column. The speed-up ranges from 1.24 (HEAP) to 2.48 (LLL8). The mean and variance of the

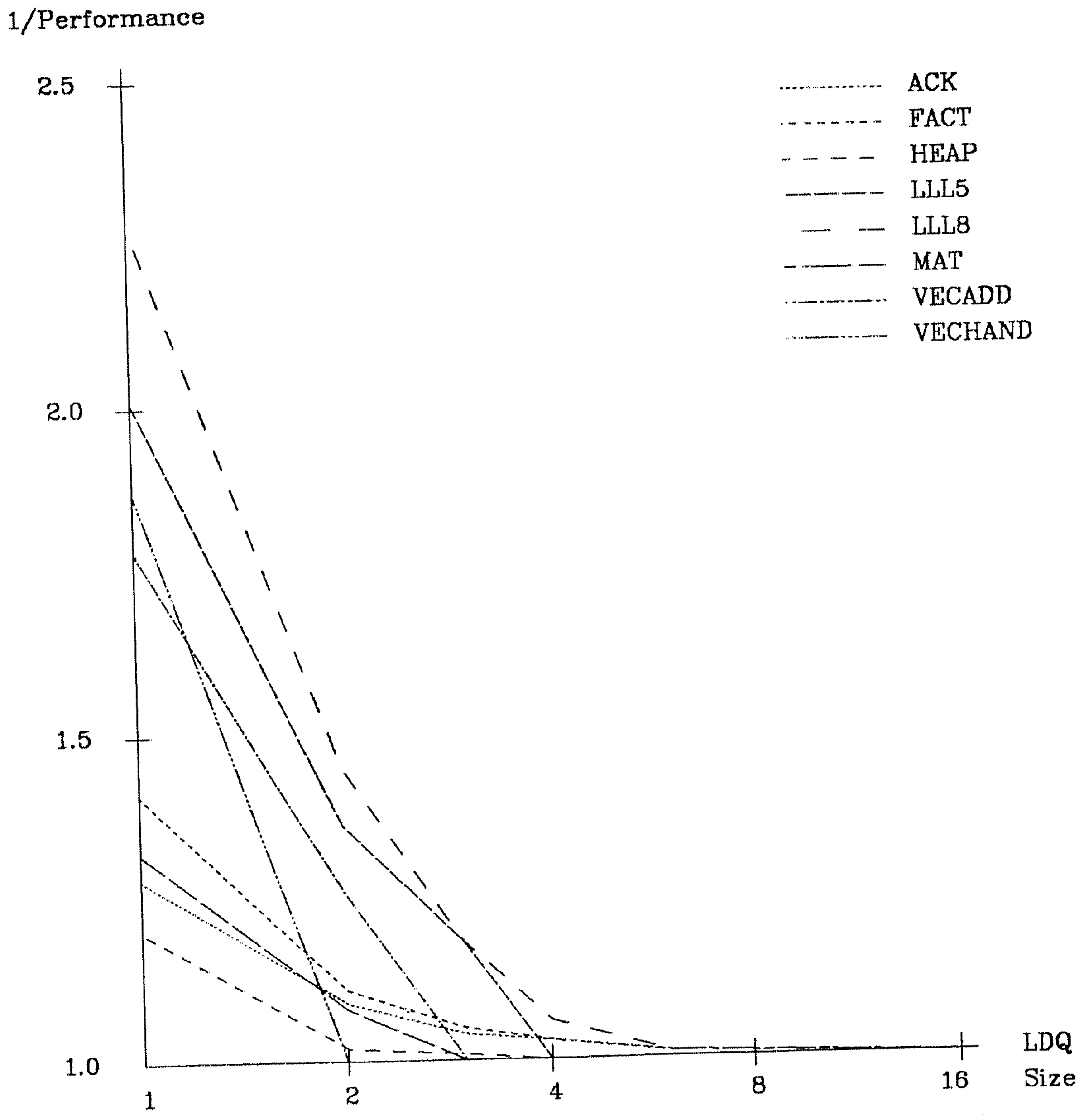

Fig. 2. Relative Performance for Different LDQ Sizes.

speed-up are 1.74 and 0.41, respectively. In table 2, we show the relative speed-up owing to the

LDQ alone. The *load distance* is the number of instruction parcels between the instruction that loads the operand and the instruction that takes that particular operand out of the LDQ. The average load distance is also included in table 2. In table 3, we show the relative performance of different LDQ sizes. In table 4, we illustrate the maximum number of entries in the LDQ at any given time. In table 5, we compare effectiveness of the PBR instruction. In table 6, we list the dynamic branch count distribution.

From this experiment, we make the following observations.

(1) Comparing table 1 and table 2, we conclude that most of the speed-up achieved is due to the LDQ.

(2) From table 3, the performance gains, due to the LDQ, level off when the LDQ size is about 4. (This, of course, is dependent on memory access time.) The relative performance of these programs for different LDQ size is shown in Fig. 2, where the y-axis is the reciprocal of the relative performance with respect to an infinite LDQ.

(3) From table 5, the performance gain due to the PBR instruction alone is not as good as that due to the LDQ alone. We offer the following reasons: (a) For a computer with elemental instructions, branch instructions occur less frequently than load instructions. (b) Since a relatively large instruction cache is used, it takes only a few cycles to complete a branch instruction even if the branch count field is zero. (c) There are not always enough instructions to follow the PBR instruction if we do the assembly language level code scheduling within a basic block. Other possibilities of doing code scheduling are discussed in section 6.

(4) From table 6, the average branch count for all the benchmark programs is 4.0 with variance 1.43. The number of instructions that can be moved after a branch instruction depends on (a) the algorithm(s) used to schedule the code; and (b) the nature of the instruction set. From table 5, the speed-up owing to PBR alone is about 10%. Though our algorithm is not as aggressive as that used with MIPS [GrHe82], our results are comparable to theirs. We have shown that the generalized delayed branch is worthwhile for a reduced instruction set computer

such as PIPE. Simple algorithms, such as the code scheduler described in this paper, can be used to take advantage of the generalized delayed branch. Exotic algorithms, such as software pipelining, may utilize the PBR instruction even better.

7.2. Software Pipelining

Supercomputer performance reported [Worl84] [RiSc84] on the Livermore loops [McMa72] indicates that often the worst performance is on loop 11 (first sum). Loop 11 is relatively hard to vectorize by current vectorizing compilers because recurrence is not well supported by the underlying hardware. It takes 15 and 8 clock periods to execute one iteration of the scheduled code and the software pipelined code, respectively. The speed-up, due to software pipelining, is 1.88.

Loop 13 (2-d particle pusher) is also hard to vectorize. The code scheduling, however, is almost as effective as software pipelining because of its relative large loop body (see section 6). If the memory access time were extremely long, software pipelining on loop 13 would be more effective.

8. Conclusions

We have demonstrated the feasibility of using the LDQ to reduce the impact of memory delay and using the PBR instruction as a generalized delayed jump. A simple code scheduler is capable of reordering the compiled code to take advantage of these special features automatically. Software pipelining can take advantage of the aforementioned features even further (in particular, the PBR instruction). The degree of prefetching (*i.e.*, the number of prefetches across the loop boundary) is determined by the execution times of different pipes in a processor. Our scheduling methods are applicable to most register-register pipelined architectures by simply changing the cost table which shows the execution times and issue conditions of instructions. The scheduling methods described in this paper will be less effective for memory-memory pipelined processors. Incidentally, but not accidentally, register-register architectures are prevalent among the scalar mode of most high performance computers (*e.g.*, Cray-1[Russ78], Cyber 205[Linc82], VP-200[MiUc83], S-810[NaIn84]) for control simplicity and other reasons. Even in the 360/370 family, where upward compatibility is critical, the internal organization of some high-end pipelined processors are of register-register form (*e.g.*, the floating point unit of the 360/91[AnST67]). Thus, we believe that queues and PBR instructions are compatible with super-computers and that the scheduling methods described in this paper are effective in utilizing special features available in PIPE.

The scheduling methods described above assume all results computed by instructions in a basic block are needed at the end of the basic block. This is always true for a branch instruction where the instructions from the branch target cannot be issued until the transfer control (the XBR point) completes (for a successful branch). This assumption leads to the fact that all cumulative costs have non-negative values. It is possible to consider the uses of an object in all successor basic blocks. If an object A will not be used, in all successor blocks, until n clock periods later and it takes m clock periods to compute A, the instruction that computes A would have a cumulative cost of $m-n$, rather than m. If $n > m$, the cumulative cost can have a negative value. This modified cumulative cost reflects the urgency of objects more than do the original cumulative cost. We are looking into the effects of this modification.

One possible extension of the LDQ is that *all* registers be implemented as queues. These queue registers can also be used as vector registers. Thus, another extension is to design a vector mode of PIPE which is compatible with the current PIPE design principle (*i.e.*, simple issue conditions). The performance implications of these extensions are under investigation.

REFERENCES

[AhU177] Alfred V. Aho, and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.

[Amda67] G. Amdahl, "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," *Spring Joint Computer Conference*, pp. 483-485, 1967.

[AnST67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IMB System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM J. Res. & Dev.*, *11*, 1, pp. 8-24, January, 1967.

[Buch83] Ingrid Y. Bucher, "The Computational Speed of SuperComputers," *Proceedings, ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 151-165, August, 1983.

[Char81] Alan E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural

Design of the AP-120B/FPS-164 Family," *IEEE Computer 14*, 9, pp. 18-27, September 1981.

[CGKP83] Gary L. Craig, James R. Goodman, Randy H. Katz, Andrew R. Pleszkun, Kishore Ramachandran, John Sayah, and James E. Smith, "PIPE: A High Performance VLSI Processor Implementation," *University of Wisconsin-Madison Computer Sciences Department Tech. Report # 513*, September 1983.

[Cray82] Cray Research, Inc. *Cray-1 Computer Systems* S Series Mainframe Reference Manual (HR-0029)

[DoJi79] J.J. Dongarra, and A.R. Jinds, "Unrolling Loops in Fortran," *Software - Practice and Experience 9*, pp. 219-226, 1979.

[FERN84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proceedings, ACM SIGPLAN'84 Symposium on Compiler Construction*, pp. 7-47, June, 1984.

[Flyn72] Michael J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput. Vol. C-21*, No. 9, pp. 948-960, September, 1972.

[GHLP85] James R. Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P.B. Schechter, and Honesty C. Young, "PIPE: A Decoupled Architecture for VLSI," submitted to *The 12th International Symposium on Computer Architecture*, June, 1985.

[GrHe82] Thomas R. Gross, and John L. Hennessy, ``Optimizing Delayed Branches``, *15th Annual Workshop on Microprogramming*, pp. 114-120, Oct. 1982.

[HJBG81] John Hennessy, Norman Jouppi, Forest Baskett, and John Gill, "MIPS: A VLSI Processor Architecture," *Tech. Report No. 223, Computer Systems Laboratory, Stanford University*, November 1981.

[HeGr83] John Hennessy, and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM TOPLAS 5*, 3, pp. 422-448, July 1983.

[HuSU77] H.B. Hunt, T.G. Szymanski and J.D. Ullman, "Operations on Sparse Relations," *Comm. of ACM 20*, 2, pp. 171-176, March 1977.

[HwBr84] Kai Hwang, and Faye A. Briggs, *Compute Architecture and Parallel Processing*, McGraw-Hill Book Company, 1984.

[Knut73] Donald E. Knuth, *The Art of Computer Programming, Vol 1*, 2nd Ed., Addison-Wesley Publishing Company, pp. 258-268, October, 1973.

[Knut73] Donald E. Knuth, *The Art of Computer Programming, Volume 2: Sorting and Searching*, 2nd printing, Addison-Wesley Publishing Company, pp. 451-471, 1973.

[Kogg81] Peter M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, New York, 1981.

[KKLW80] David J. Kuck, Robert H. Kuhn, Bruce Leasure, and Michael Wolfe, "The Structure of an Advanced Retargetable Vectorizer," *Proceedings of COMPSAC*, pp. 709-715, October, 1980.

[Lee80] Ruby Bei-Loh Lee "Empirical Results on the Speed, Efficiency, Redundancy and Quality of Parallel Computations," *Parallel Processing Conf.* pp. 91-100, 1980.

[LeSm84] Johnny K.-F. Lee and Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer 17*, 1, pp. 6-22, January 1984.

[McMa72] F.H. McMahon, "Fortran CPU performance Analysis", *Lawrence Livermore Laboratories*, 1972.

[MiUc83] Kenichi Miura, and Keiichiro Uchida, "FACOM Vector Processor System:VP-100/VP-200," *Proceedings of NATO Advanced Research Workshop on High Speed Computing*, West Germany, June 1983, reprinted in *IEEE, Tutorial Supercomputers: Design and Applications*, Kai Hwang (Ed.), pp. 59-73, August, 1984.

[NaIn84] Shigeo Nagashima, and Yasuhiro Inagami, "Design Consideration for a High-Speed Vector Processor: The HITACHI S-810," *IEEE, ICCD'84*, pp. 238-243, October, 1984.

[PaSe81] David A. Patterson, and Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings of the Eight Annual Symposium Computer Architecture*, pp. 443-457, April 1981.

[PaSe82] David A. Patterson, and Carlo H. Sequin, ``A VLSI RISC,'' *IEEE Computer, 15*, 9, pp. 8-21, September 1982.

[Radi82] George Radin, "The 801 Minicomputer," *ASPLOS SIGARCH Computer News, 10* March 1982. Reprinted in *The IBM Journal of Research and Development, 27*, 3, pp. 39-47, May 1983.

[RiSc84] John P. Riganati, and Paul B. Schneck, "Supercomputing," *IEEE Computer, 17*, 10, pp. 97-113, 1984.

[RiFo72] Edward M Riseman and Caxton C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Trans. Comput. Vol. C-21*, No. 12, pp. 1405-1411, December, 1972.

[Russ78] Richard M. Russell, "The Cray-1 Computer System," *CACM, 21*, 1, pp. 63-72, January, 1978.

[Smit81] James E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Symp. Computer Architecture*, pp. 135-142, May, 1981.

[SPKG83] James. E. Smith, Andrew R. Pleszkun, Randy H. Katz, and James R. Goodman, "PIPE: A High Performance VLSI Architecture," *Proceeding, IEEE International Workshop on Computer Systems Organization*, pp. 131-138, March 1983. Also available as *University of Wisconsin-Madison Computer Sciences Department Tech. Report # 512*, September, 1983.

[Thor70] J. E. Thornton, *Design of a Computer, The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill. 1970.

[Weis84] Shlomo Weiss, "Very High Performance Scalar Processing," *Tech. Report, ECE-84-22, Electrical and Computer Engineering Department*, University of Wisconsin-Madison, September, 1984.

[Worl84] Jack Worlton, "Understanding Supercomputer Benchmarks," *Datamation*, pp. 121-130, September, 1984.

[YoGo84a] Honesty C. Young and James R. Goodman, "A Simulation Study of Architectural Data Queues and Prepare-to-branch Instruction," *Proceedings, IEEE International Conference on Computer Design*, pp. 544-549, October, 1984.

[YoGo84b] Honesty C. Young and James R. Goodman, "Software Pipelining for a Pipelined Computer," *Proceedings, International Computer Symposium*, December, 1984.