Optimization on Instruction Reorganization

Feipei Lai Hung-Chang Lee Chun-Luh Lee

Dept. of Electrical Engineering
&
Dept. of Computer Science
National Taiwan University
Taipei, Taiwan, ROC.

Abstract

A pipelined processor increases its performance by partitioning an instruction into several separate operation steps. Several instructions can be executed in the pipeline in different pipe stages at the same time. Since the overlapped execution of instructions, the result of an instruction may be attempted to be used before it is available.

One way to solve this problem is to schedule instructions at compiler time, thus the codes generated will be free from interlocks. The scheduling algorithm presented by [Hen 83, Gro 83] had significantly reduced the pipeline interlocks. With some modifications to distinguish the conflict condition, the algorithm will do better works at the same cost.

1 Introduction

A pipelined processor divides each instruction into a sequence of basic operations, each basic operation can be executed by a special functional unit that operates concurrently with other units in the pipeline. While in a nonpipelined processor, it must complete all these steps before the next instruction can be issued. Suppose S is the pipeline stage number and T is the time units that a nonpipelined processor needed to complete an instruction. For simplicity, we assume the turn-around time of an instruction in two processors is the same. Since a pipelined processor performs overlapped operations on instructions to exploit temporal parallelism, it can complete an instruction in every T/S time units. The ideal speedup over a nonpipelined processor is S (as illustrated in Figure 1 and 2). However, they are not always the case, due to memory conflict, data dependence, conditional branches and interrupts, this speedup can seldom be achieved ([Lil 88]). There are several strategies to overcome these degradation factors in either software or hardware.

Most of the pipelined processors provide hardware circuits that act as a bypass channel between pipe stages to lessen the effects caused by dependences among operands. The circuit that performs this bypass service is known as *internal forwarding*.

Some high speed computer systems also implement complex circuits to runtime check dependencies in instructions, delaying those instructions that their operands are not avail-

able and continuing those instructions that are not suffered by the hazardous data. The floating point unit of IBM 360/91 [Tom 67] and CDC 6600 [Tho 64] are examples of the pipelined systems that implement this dynamically scheduling mechanism to change the execution order.

Figure 1: Executing processes on a nonpipelined processor.

Figure 2: Executing processes on a pipelined processor.

The same scheduling process can also be implemented in software. Dependence relations among instructions are checked at compiling time. The codes generated are thus free from pipeline interlocks. Researches on code reorganization in compiling time are still sparse, though they are very important for pipelined architectures, VLIW (Very Long Instruction Word) machines and multiprocessor systems. In those researches, [Allan 86] developed a phase coupling method for horizontal microcode generation. [Gro 83] presented an algorithm with heuristic cost function to reorganize instructions. This research was as part of the MIPS project developed at Stanford University. It has a worst case complexity $O(n^4)$. [Gib 86] proposed a heuristic algorithm with worst case complexity $O(n^2)$. These differences in efficiency are majorly caused by the dependent relations which they treated in building DAG (Dependent Acyclic Graph). Kuck and his fellows at University of Illinois had defined three dependence relations [Kuc 78, Tow 76].

In all these dependence relations, only the *true dependence* would cause timing hazard. *Antidependence* is produced by reusing the variable. The purpose to write a program with this dependence is to save storage space. This dependence may be removed by renaming the variable if we have enough resources. In the last case, if there is no *true*

dependence or *antidependence* between the statements which results in output dependence, then this must be a dead code and could be removed so the dependence disappears. Normally, in data flow analysis the only dependence relation we take into account is *true dependence*. Since the registers in a processor are limited and scarce, reusing a register when its value is not referenced any more is unavoidable. This is the major source where antidependence comes. In most RISC architectures, all of the operations must be executed in registers, so the *antidependence* may appear frequently in assembly codes.

[Gro 83] built an edge in DAG only when there is a true dependence occurs, while [Gib 86] built a dependence edge whenever there is true dependence, output dependence or antidependence. These differences make [Gib 86]'s algorithm more efficient but less optimal. The following example shows their differences (Figure 3).

Figure 3: The difference between [Gib 86]'s and [Gro 83]'s dags.

2 The Issues

The previous scheduling algorithm presented by [Hen 83, Gro 83] has been proved to be an excellent work on instruction reorganization process. With some modifications, we have got a new algorithm which will generate even better codes sequences at the same cost. We will introduce that algorithm in the following sections.

3 Assumptions

In our target machine MARS (Multiprocessor Architecture Reconciling Symbolic with numerical processing) the pipeline is divided into five stages. Those are IF, ID, ALU, MEM and WB [MARS 89]. Each instruction comes into the pipeline at the IF stage and goes out the pipeline at WB stage. Normally this scheme will cause timing hazard when two instructions are data dependent and the separation between them is less than three cycles.

Because data dependence appears in programs so frequent, a hardware internal forwarding mechanism is implemented to resolve the data dependence among register operands. If any two succeeding instructions have the same operands bypassed, the former operand is overridden by the updated value. This situation may occur when an instruction is squashed by an earlier branch. The strategy insures the squashed operand not bypass to a successful operation (Figure 4). Dependent operands in registers are thus resolved. Most dependences in programs will therefore not cause any timing hazard problem. The only resource that would result in timing hazard is the "load" instruction. It computes the address of operand in the ALU phase, then sends address out. And the operand is returned at the end of the MEM cycle. What the optimization focuses on is to resolve the hazard caused by load instructions.

Figure 4: Override the older data for bypassing.

4 The Definitions

Before we come into the optimization algorithm, there are some definitions that will make the problem more clear. These definitions are digested from Thomas Gross [Hen 83, Gro 83]. Here are the definitions.

Definition 1 (Parallel) Two nodes are parallel if they have neither ancestors nor children in common. A node n is parallel to a directed acyclic graph (dag) D if all the nodes in D are parallel to n .

Definition 2 (Conflict) Two machine instructions conflict if each has a destination that is a source for the other instruction, or if they have the same destination.

Definition 3 (Blocked) A code reorganizer is blocked if it reaches a point where the only remaining choices of instructions conflict.

Definition 4 (Covered & Fully Covered) In generating code for a dag D , the covered set of nodes is the set of nodes for which instructions have been generated. A node n is fully covered if n is in the covered set of nodes and there are no references in the uncovered set of nodes to the node n . A node whose register value is alive at the end of the basic block is never fully covered.

A node n is fully covered if and only if all parents of n are covered. If a register holds a value needed by the node or needed by a computation that must follow the node, then we say that the register is live.

Definition 5 (Live) If N is a set of covered nodes, then $r \in \text{Live}(N)$ if r is the destination of a node in N and that node is not fully covered in N .

Definition 6 (Cover Set) Given a forest of dags D and a node n . Assume that n is scheduled. Let N be the set of nodes that must be covered to fully cover node n . Then the set $\text{Cover}(n)$ contains the resources required to fully cover node n . That is, $\text{Cover}(n) = \{ r \mid r = \text{dest}(n') \text{ for all } n' \in N \}$.

Definition 7 (Safe Position) Given a forest of dags D with covered nodes C and an uncovered node n . Node n is a safe position for resource r in D if

- $r \neq \text{dest}(n)$, or
- $r = \text{dest}(n)$, then for $\forall i$ parallel to n :
 $r \notin \text{Live}(i) \cup \text{Cover}(i)$, or $\text{dest}(i) \notin \text{dest}(n) \cup \text{Cover}(n)$

Definition 8 (Inhibited) A node n is called inhibited if n or a descendant of n is not a safe position for $\text{dest}(n)$.

Definition 9 (Safe Path) Given a dag D with a set of covered nodes C and a starting node i , a safe path is a set $SP(C, i)$ such that:

- if i is not a safe position, then $SP(C, i) = \emptyset$; otherwise

- if i is a safe position then

1. if there is a node n' with $\text{dest}(n') = \text{dest}(i)$ and n' is neither fully covered nor inhibited, then $SP(C, i) = \{ n \mid n \text{ direct ancestor of } i \} \cup SP(C, j) \forall j \text{ uncovered child of a direct ancestor}$.
2. if all nodes n' , with $\text{dest}(n') = \text{dest}(i)$, are fully covered or inhibited, then $SP(C, i) = \{i\}$.

Definition 10 (Overlap) Given a set of covered nodes C , two safe paths for nodes n_1 and n_2 overlap if there is a node with a destination of $\text{dest}(n_2)$ in $[SP(C, n_1) \setminus SP(C, n_2)]$, or a node with a destination of $\text{dest}(n_1)$ in $[SP(C, n_2) \setminus SP(C, n_1)]$.

5 Heuristic Solution

The scope in which our reorganizer schedules instructions is limited within a basic block. Since the reorganizer operates on both compiler generated and hand-script assembly codes. The latter may have been written without any knowledge about pipeline conflicts and thus will cause unpredictable results if they are not scheduled by the reorganizer. The reorganization process is as follows:

1. Read in a basic block and create a dag.
2. Determine the set of instructions that can be issued.
3. Delete any instruction that cannot be started at this moment.
4. Select (heuristically) an instruction from the set.

During the reorganization process, some problems should be treated carefully. They are: (1) The live information about registers among basic blocks must be provided before the reorganizer starts to reorder instructions. (2) Branch instruction cannot be moved, it must remain as the last instruction of the basic block. (3) Instructions which could alter the content of the same memory location have to be scheduled strictly serialized. Here lists the timing optimization algorithm:

Assume $D_1, D_2, \dots, D_n$ are parallel dags, with covered sets $C_1, C_2, \dots, C_n$.

1. Compute the set of nodes which has no child or all their children are covered, such that:
 - (a) if $i \in D_j \setminus C_j$, all children k of $i \subseteq C_j$, and $\forall k$, i is not interlocked by k (because of timing hazard), then put i into the *Ready* set.
 - (b) Starting with i , find the minimal safe path from C_j , $SP(C_1 \cup \dots \cup C_n, i)$.
 - (c) If $j \in SP(C_1 \cup \dots \cup C_n, i)$ is "load" or "store", make sure that it does not access a memory location out of order. Here we must insure that each "load" instruction is executed before its content being changed. Otherwise, remove node i from

the *Ready* set.

- (d) If the safe paths of two starting nodes overlap, choose one from the starting nodes using the heuristic cost function.
2. If *Ready* set is empty, emit an *Nop*. Otherwise, choose an instruction i from *Ready* set. If in step 1.d a safe path has been eliminated, then
 - (a) Mark all the nodes in $SP(C_1 \cup \dots \cup C_n, i)$, with a number "1", which satisfy $dest(n) = dest(r) \forall r \in Ready$ set, "2" otherwise.
 - (b) The next instruction emitted must be chosen from the selected safe path until all the nodes marked "1" in the path have been scheduled.
 - (c) Now we can repeat (1) to choose another node and emit it.
3. Repeat steps (1) and (2) until all the nodes are emitted.

The heuristic cost functions which we have used in timing optimization to choose a node from two or more safe paths and overlapped paths are listed below in decreasing priority:

- choosing "load" instruction first,
- choosing the node that is farthest from root,
- choosing the one that has more parents,
- choosing "store" instruction.

In our approaches, the *dags* are constructed in the same way as [Gro 83]. The major difference between our heuristic algorithm and [Gro 83] is that we partition the set *safe paths* into two subsets. Those nodes which would cause conflict condition are marked with a number "1", these nodes should be emitted before the other *safe path* can be scheduled. While those nodes which would not cause conflict problem are marked by a number "2". Only the nodes marked with the number "1" must be completely scheduled before another *safe path* can be scheduled. This feature provides us another chance to intermix nodes in overlapped *safe path*, and thus it would generate better codes than those generated using the algorithm proposed by [Gro 83].

However, nothing is gained without any sacrifice. The additional cost we pay is the marking procedure, which takes action only when an overlapped *safe paths* occurs. As a result, the total cost of the marking procedure is that multiplication of the number of situation that an overlapped *safe paths* happens, the number of node in *ready set* whenever an overlapped *safe path* occurs, and the number node in the selected *safe path*. All these three values are bounded by the total nodes in the DAG, i.e., N . We can, therefore, obtain the upper bound of $O(n^3)$ for the additional cost of the marking procedure. This bound is extremely pessimistic, as the bounds for the three values include several worst case assumptions. For example, a *ready set* with the upper bound of N assume all the node is ready. However, marking the node in the *safe path* is trivial, since there will not be any node left. As a whole, the time complexity of our algorithm

is that same as that of [Gro 83], i.e., $O(n^4)$.

Consider the example (Figure 5), these codes are frequently used in a loop to calculate the summation of an array. The unoptimized solution and reordered codes are as follows:

unoptimized solution: 1 nop 2 3 4 nop 4 5 6 7 8
after optimization: 1 4 2 3 7 5 6 8

$s = s + a[i]$
 $i = i + 1$

1	load	4[R15],	R2	
2	sll	R3,	R2,	#1
3	load	R3[R7],	R4	
4	load	8[R15],	R1	
5	add	R1,	R4,	R1
6	store	R1,	8[R15]	
7	add	R2,	R2,	#1
8	store	R2,	4[R15]	

Figure 5: Summation of an array (statements in a loop).

The heuristic cost functions are listed in decreasing priority. The motivation for which these cost functions are defined is that: (1) Since all the timing hazards are resulted from "load" instructions, so we scheduled it first. If there is another instruction could be emitted immediately, then we would have solved this hazard. (2) Each *dag* has a longest path, the longest path in all the *dags* is the minimal time to complete these instructions not to mention what parallelism can be achieved. (3) Choose the node that has more parents because that it may result in more nodes to be *ready*. (4) Increase the "store" instructions priority is because that they

may incur alias problem to constrain "load" to be emitted.

The following example shows how our algorithm works. Consider those instructions listed in (Figure 6), at the beginning, *Ready* set contains node 1 and node 3, $SP(0,1) = \{1,2\}$, and $SP(0,3) = \{3,4,5\}$. Since $dest(n_1) = R1 \in SP(0,3) \setminus SP(0,1)$ and $dest(n_3) = R2 \in SP(0,1) \setminus SP(0,3)$, these two safe paths overlap. According to [Gro 83], these two safe paths must be scheduled sequentially. That is, there would be two idle cycles in the generated codes. In our algorithm, we can issue them with only one idle cycle. The following example shows how our algorithm works. According to Algorithm 2.(a), node 5 is marked "1" and node 4 is marked "2". Since node 1 and 3 are "load" instructions and their distance from root are the same, we choose node 3 because its parents number is larger than that of node 1. After node 3 is emitted, the next instruction to be emitted is an *Nop*. This is because the marked node 5 is interlocked by node 3. When node 5 is emitted, the next node scheduled is node "1", then node "4", and node "2". The instructions will be scheduled in the following sequence.

[Gro 83]: 1 nop 2 3 nop 4 5
Our optimization: 3 nop 5 1 4 2

Figure 6: A simple reorganization example.

6 Results & Conclusion

The scheduling algorithm proposed by Hennessy and Gross [Hen 83, Gro 83] was a milestone in software reorganization. Although it has a worst case runtime complexity $O(n^4)$. Where n is the number of nodes in a *dag*. But according to [Gro 83], the average number of instructions ready is close to 2, and about 1.22 safe path are computed per node. This means that in practical the algorithm is quit efficient.

We modified the algorithm presented by [Hen 83, Gro 83]. At the same optimization cost, we have got another chance to schedule instructions over the previous work. With this slightly modification, the algorithm will produce higher quality codes than before. The benchmark programs are listed in table 1.

Sub program	source lines	default nops	optimize nops	removed nops	percentage rem/def
acker	26	16	2	14	87.50%
bubble	54	31	8	23	74.19%
fib	20	12	1	11	91.67%
hanoi	75	51	5	46	90.20%
knight	88	37	7	30	81.08%
perm	65	33	9	24	72.73%
preorder	17	10	1	9	90.00%
qsort	84	54	11	43	79.63%
queue	84	43	14	29	67.44%
shell	84	49	10	39	79.59%

Table 1: Benchmark results

References

- [Aho 77] A. V. Aho, and Ullman, J. D. *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
- [Allan 86] Vicki H. Allan and Robert A. Mueller, "Phase Coupling for Horizontal microcode Generation," *Proc. 21st Annual Workshop Microprogramming and Microarchitecture, MICRO-21*, ACM SIGMICRO, Vol. 18, No. 4, Feb. 1988. pp. 24-28.
- [Gib 86] Phillip B. Gibbons and Steven S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," *Proc. of ACM Symposium on Compiler Construction*, ACM, Palo Alto, June 1986. pp. 11-16.
- [Gro 83] Thomas Gross, *Code optimization of Pipeline Constraints*, Ph.D. dissertation, Department of Electrical Engineering and Computer Science. Stanford University. December, 1983.
- [Hen 83] J. Hennessy and T. Gross, "Postpass code Optimization of Pipeline Constraints," *ACM, Transaction on Programming Language and Operating Systems*, Vol. 5, No. 3, July 1983, pp. 422-488.

- [Kuc 78] D. J. Kuck, *The Structures of Computers and Computations*, Vol. 1, Wiley, New York, 1978.
- [Lil 88] David J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," *IEEE Computer*, July, 1988, pp. 47-55.
- [MARS 89] G-S Jang, et al., "MARS-Multiprocessor Architecture Reconciling Symbolic with Numerical Processing," *Int'l Symp. VLSI Technology, Systems, and Applications*, May 1989, pp. 365-370.
- [Tho 64] J. E. Thornton, Parallel Operation in the Control Data 6600, *Proc. Fall Joint Comp. Conf.*, Part 2, Vol. 26, 1964, pp. 33-40.
- [Tom 67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Res. and Dev.*, Vol. 11, No. 1, Jan. 1967, pp. 25-33.
- [Tow 76] R. A. Towlf, *Control and data dependence for program transformations*, Ph.D. dissertation, Report, 76-788, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1976.