COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD, CA 94305-4055

Code Generation and Reorganization in the Presence of Pipeline Constraints

John Hennessy and Thomas Gross

Technical Report No. 224

November 1981

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680. Thomas Gross is supported by an IBM Graduate Fellowship.

Code Generation and Reorganization in the Presence of Pipeline Constraints

John Hennessy and Thomas Gross

Technical Report No. 224

November 1981

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

Pipeline interlocks are used in a pipelined architecture to prevent the execution of a machine instruction before its operands are available. An alternative to this complex piece of hardware is to rearrange the instructions at compile-time to avoid pipeline interlocks. This problem, called code reorganization, is studied. The basic problem of reorganization of machine level instructions at compile-time is shown to be NP-complete. A heuristic algorithm is proposed and its properties and effectiveness are explored. The impact of code reorganization techniques on the rest of a compiler system are discussed.

Key Words and Phrases: Code generation, pipelining, interlocks, instruction reordering, code optimization, register allocation, microprogramming

A version of this report will appear in the Proc. of the Ninth ACM Conference on Principles of Programming Languages, 1981

1 Introduction

Recent research in computer architecture centers around two major trends: the development of architectures that attempt to support high level language systems through more sophisticated instruction sets, and the design of simpler architectures that are inherently faster but may rely on more powerful compiler technology. The latter trend has several properties that make it an attractive host for high level languages and their compilers and optimizers:

1. Because the instruction set is simpler, individual instructions execute faster.

2. A compiler is not faced with the task of attempting to utilize a very sophisticated instruction that does not quite fit any particular high level construct. Besides slowing down other instructions, using these instructions is sometimes slower than using a customized sequence of simplier instructions [12].

3. Although these architectures may require more sophisticated compiler technology, the potential performance improvements to be obtained from faster machines and better compilers are substantial.

Recently, several articles have discussed the relationship between compilers, architectures and performance [16, 5]. The concept of simplified instruction sets and their benefits, both for compilers and hardware implementations, are presented in [11, 12].

The unique property of some of these experimental architectures is that they will not perform efficiently without more sophisticated software technology. This paper investigates a major problem that arises when generating code for a pipelined architecture that does not have hardware pipeline interlocks. Without hardware interlocks, naively generated code sequences will not run correctly. These interlocks must be provided in software by arranging the instructions and inserting no-ops (when necessary) to prevent undefined execution sequences. There are currently several architectures that require software imposition of certain types of interlocks [10, 6]. The absence of interlocks is also very common in micromachine architectures, and the microprogrammer must often address this problem.

1.1 Background

A pipelined processor is one in which several sequential instructions are in simultaneous execution, usually in different phases. One component of an instruction may refer to a component that is computed in an earlier instruction. Because the earlier instruction may still be executing, the value of the component may not be available. A hardware mechanism, called a pipeline interlock, prevents the latter instruction from continuing until the needed value is available.

Figure 1 shows a typical pipeline configuration. This pipe has three stages: ps_a, ps_b, ps_c. Three instructions

arc under execution at any time: instruction 1 at ps_c, instruction 2 at ps_b, and instruction 3 at ps,. A pipeline interlock would prevent instruction 2 from executing ps_b if, during ps_b, instruction 2 required a value calculated during ps_c of instruction 1 (since that value would not be available until the end of ps_c and the two pipestages (1,ps_c) and (2,ps_b) are coincident).

[1]	ps_a	ps_b	ps_c			
[2]		ps_a	ps_b	ps_c		
[3]			ps_a	ps_b	ps_c	
[4]				ps_a	ps_b	ps_c

Figure 1: Sample Pipeline

In a pipelined architecture the design of the interlock mechanism is complex and adds significantly to the hardware overhead of a high-performance processor [9, 15]. The presence of interlock checks also imposes an overhead on all instructions whether or not they arc affected by the interlocks. The elimination of this hardware will allow a simpler design. If the implementation of a processor on a single VLSI chip is attempted, simplicity and regularity can be crucial for the success of the project. Also. a simpler design allows for more regularity and eventually a smaller and faster chip.

Another reason for the interest in such an architecture is the potential speed-up. If the code can be reordered so that legal instructions arc always being executed, then no time is lost due to the resolution of interlocks, and the code will run faster than the unordered code would run on a pipclined machine with interlocks. A penalty is only paid when the code sequences have to be padded with no-ops; even in this case there may not be a time penalty involved. The elimination of the occurrence of dynamic interlocks will also speed up conventional pipelined architectures.

2 The problem

The problem faced by a code generator for such an architecture is to guarantee the correct execution of the original program, i.e. to ensure that the input-output function of the program is identical when the program is executed in a single step fashion and when it is executed without interlocks by pipclined hardware.

2.1 interlocks

Due to the overlap of the execution of instructions and the pipeline structure, the results of instruction i are not available until instruction $i+k$. An attempt by instruction $i+k$ 'with $k' < k$ to rcfcrcnce data written by instruction i will result in a delay. Interlock hardware will detect this case and block the execution of instruction $i+$ k'until the data is available. More formally, this is called a *destination-source* conflict:

Definition 1: Let *i* bc an instruction and ps be a pipestage, then register *R* is a *source field* in *(i ,ps)* if and only if instruction *i* reads thc value of register *R* during cycle *ps*. *R* is a *destination field* in *(i ,ps)* if *R* is written during thc *ps* cycle of instruction *i*.

Definition 2: A *destination-source pipeline interlock* is a constraint of the form $(i_1,ps_1)...(i_2,ps_2)$. If a register *R* is a destination field in (i_1,ps_1), then it may not be a source field in any (i',ps') which is executed after (i,ps_1) and earlier than the pipestage following (i_2,ps_2). A source-destination pipelinc interlock is the same as a destination-source interlock except that *R* is initially a source and later used as a destination.

Destination-source interlocks arc a natural result of a pipeline structure. *Source-destination* conflicts result from the possibility of interrupts or page faults. In Figure 1 the execution of $(2,ps_a)$ will normally precede thc execution of $(1,ps_c)$. But if there is a page fault when instruction 2 is fetched, instruction *1* will complete. Then the page fault will bc handled. In this case, stage $(2,ps_a)$ will not be executed before $(1,ps_c)$. Therefore, $(1,ps_c)$ should not depend on the results of $(2,ps_a)$.

Definition *3:* A *single instruction interlock* is a special case where instruction i_2 is the sequential successor of i_1.

Most interlocks result because destinations are usually results that arc computed near the end of a pipeline, and such results arc often used in the next instruction as a source. For example, consider the pipeline in Figure 1: assume that sources for alu operands arc fetched from registers during ps, and thc results are stored during ps_c. Figure 2 shows two statements and the obvious but incorrect code sequence; the value of *R1* is not available when it is used either in the first or second Add instruction. A correct code sequence, assuming single instruction interlock, is also shown.

Statements	Incorrect Code	Correct Code
A := B + c	Load B, R1	Load B, Rl
	Add C, R1	Load C, R2
A := A + E	Add E, R1	Load E, R3
	Store R1, A	Add R1,R2
		Add R2, R3
		No-Op
		Store R3, A

Figurc 2: Incorrect and correct code sequences

2.2 Possible solutions

There are two possible approaches to solving the code generation problem in the presence of noninterlocked hardwarc. First, the problem can be stated as an extension to the standard dag-based code generation problem [2]. In this form it is clear that the code generation problem for the dags with interlocks is at least as hard as the optimal code generation problem for a register-based machine (known to be NP-complctc), Furthermorc, a heuristic code gencration algorithm for dags would probably bc extremely

complex. Some of these problems can be solved by using a tree representation and ignoring the possibility of common subexpressions and the existence of multiple trees in a basic block. However, this simplified approach may result in unacceptable code quality, particularly since machine instructions belonging to more than one statement can not be in termixed to avoid in terlocks.

An alternative approach is to reorganize the code to meet the interlock requirements in a postpass following code generation. The most important advantage of this approach is that it can be applied both to code output from a compiler and to hand-written assembly language code. The absence of hardware interlocks makes it extraordinarily difficult to generate correct programs in assembly language, and a software reorganization tool is needed to support such programming. The other advantage is the decomposition of the problem into two simpler problems, although this means that the final result may not be optimal. This is probably not serious since the goal of optimal code is not obtainable with a practical algorithm independent of the two-step approach. Very slow near-optimal algorithms may be unsuitable because they must be used during each compilation, not just optimizing runs.

2.3 Related problems

This problem and our approach are related to some of the work done in microcode optimization [14, 4], but are also significantly different. Local microcode optimization begins with a correct vertical microcode sequence and improves the code by compacting the vertical microoperations in horizontal microinstructions. Two operations can be compacted into a single instruction if data dependencies are preserved, and the operations do not overlap either in resource utilization or encoding space in the microinstruction.

A primary characteristic of the horizontal microcode optimization problem is the assumption that resource utilization by an operation is for a fixed period independent of the operation's context. Microcode optimization also assumes that the order of memory accesses cannot be altered from the original version of the microcode.

In contrast, the pipeline reorganization problem concerns interlocks whose effect is a dynamic property. The context of a particular instruction determines whether or not that instruction is legal in its current position. Also, reorganization utilizes interchanges of loads and, to a lesser extent, stores, as a primary optimization technique.

3 Problem representation

The problem of code reorganization to meet pipeline constraints requires a reprcscntation of the machine level code that is generated without the effects of the interlocks, and a reprcscntation of the machine's interlocks. Most pipeline interlocks deal with access to register contents and we will consider only this type of interlock. Because of aliasing problems, interlocks involving memory are not detectable except dynamically at runtime. Most architectures maintain strict sequential access to memory (at least for stores) and eliminate the need for interlocks on memory. We assume that a simple function *WillInterlock*, given a set of parameters describing the two instructions and their separation, will return true if an interlock exists.

3.1 A data structure for machine code

An extended labeled dag [1], called a machine-level dag, is used to represent the machine code emitted by the compiler. Labels within nodes specify the value that the node has on entry to the basic block, and external labels specify register destinations (as well as operations). Memory stores are represented as nodes in the dag. A set of instructions and the resulting dag are shown in Figure 3. The dag data structure is extended to link the references of one register to subsequent modifications. This link represents the ordering that is required to maintain the semantics of the machine language code ordering. If node n modifies a register, that register is denoted by dest(n). A basic block (in the machine language) is translated into a forest of dags. Unconnected dags (or subdags) are called *parallel*.

Machine code

```
Add #1, R1, R2
St  R2, 4(sp)
Add R3, R2, R2
```

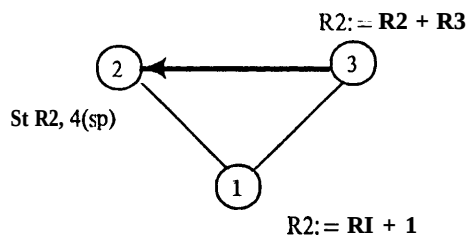


Figure 3: Part of a dag

Proof: First, WC show that the modified and original reorganization problems are equivalent which shows that the reorganization problem is NP-hard.

Suppose we had a modified organization problem that WC wished to solve. If the no-op nodes are removed and the reorganization problem is solved, the solution will also be the solution for the modified problem.

Similarly, if we had a reorganizations problem to solve, an interlock-free dag could be constructed from the initial input. The solution to the modified problem with the interlock-free dag is also the solution to the original problem.

The reorganization problem is in NP. An algorithm for the problem is to nondeterministically evaluate all possible orderings of the dag. There are only a finite number of orderings, since WillInterlock is false for every pair of instructions and some amount of instruction separation. The evaluation of each ordering (i.e. whether it is legal and how many no-ops it has) can be done in polynomial time. The legality checks are similar to those needed to check for correct code generation from a dag, and the interlock checks are trivial. The algorithm then selects the ordering that is shortest legal ordering. ∎

4 A heuristic solution

As this reorganization process will be part of each compilation and as generating optimal code is very expensive, we will concentrate on "good" solutions rather than on optimal ones. The basic strategy is as follows:

1. Read in a basic block and create a machine-level dag.
2. At any point determine sets of instructions that can be generated.
3. Eliminate any sets that cannot be started immediately.
4. Choose among the sets left.

The same register can be used in different parallel dags in a basic block (or in parallel subdags within a single dag). Because of this it is possible to emit code for the nodes in a dag such that two parallel parts of one or more dags will block each other.

Definition 10: Two machine instructions *conflict* if each has a destination that is a source for the other instruction.

Definition 11: A code reorganizer is *blocked* if it reaches a point where the only remaining choices of instructions conflict.

Because of the potential for blocking, when selecting the next instruction it is not sufficient for the reorganizer to look only at the instructions which are ready to be scheduled. Instead, the reorganizer must look ahead to determine that the nodes being selected will not lead to a deadlock situation. In Figure 4 the transformation

3.2 The complexity of reorganization

Using the machine level dag representation, we can state the reorganization problem more formally.

Definition 4: Given a machine level dag, D, with a valid register assignment, and a function *WillInterlock*, the *reorganization problem* is to generate a sequence of instructions using a minimal number of no-ops such that:

- the instruction sequence is a legal evaluation of the dag, and
- for all instructions i_1, i_2 in the sequence: WillInterlock (i_1,i_2,n), where n is the instruction separation between i_1 and i_2 is false.

We will show that the reorganization problem is NP-hard by showing that a modified version of the problem contains the register assignment feasibility problem.

Definition 5: Let D be a machine-level dag with a valid register assignment. An *interlock-free dag*, D', is constructed from D by inserting *m* no-op nodes between every pair of connected nodes in D. The value *m* is one greater than the maximum value for which WillInterlock can be true.

Definition 6: The *modified reorganization problem* is: given an interlock-free dag containing no-ops, generate code from that dag using as few of the no-ops as possible. Note that the original register assignment must be kept intact.

Later we will show that the original and modified reorganization problems are equivalent. First, we show that the register assignment feasibility problem can be imbeddcd in the modified reorganization problem.

Definition *7:* The *register assignment feasibility* problem [13] is: given a dag and a register assignment for the nodes of the dag, is there an evaluation order of the dag for which the register assignment is valid (i.e. the instruction sequence does not store into a register before all uses of the value in that register have been employed)?

Lemma 8: The modified reorganization problem is NP-hard.

Proof: By reduction of the register assignment feasibility problem which is NP-complete.

Let D be a dag whose register assignment we wish to test, construct D' by inserting a no-op between every pair of connected nodes in D. Since the no-op nodes arc not labelled with a register name, D' obviously has a legal evaluation using the register assignment supplied by D.

Claim: The initial assignment of D is feasible iff the solution to the modified reorganization problem, using D' and WillInterlock = false for all inputs, is a code sequence with no no-ops.

If *case:* When the solution to the modified reorganization problem with WillInterlock = false contains no no-ops, the resulting instruction sequence shows a legal evaluation order for the dag with the initial register assignment.

Only if case: If the initial assignment is valid and WillInterlock = false, then the no-ops are needed only to prevent overlapping uses of a register. Thus, the no-ops are needed only if the initial assignment is not feasible. ∎

Theorem 9: The reorganization problem is NP-complete.

```
1   Ld A, R1
2   Add #1, R1, R4
3   Ld B, R4
4   Sub #1, R4, R1
```

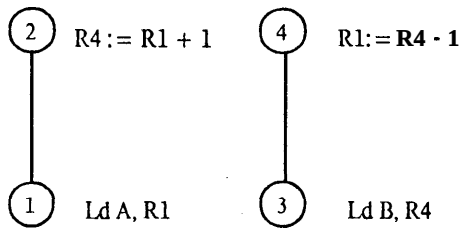


Figure 4: Dag with potential deadlock

will end in a deadlock if the schedule starts with instruction 1 followed by instruction 3. To capture this problem, we introduce the notion of a *safe set* of nodes in a dag.

Def'inition 12: 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 covered and there are no references in the uncovered portion of *D* to the node *n*.

Definition 13: Given a dag D, a *safeposition* in D is a set of nodes, S, s.t. $n \in S$ iff:

- dest(n) is not a label in any parallel dag D', or
- if dest(n) is $\in$ D', then n is fully covered.

Our algorithm does not back up; therefore, we have to prevent deadlock situations. A safe position is a set of nodes, S, in the dag such that: code has been generated for all the nodes in S, and the nodes in S do not affect the generation of code for the remaining nodes in the dag but not in S.

Definition 14: Given a dag D with a covered subset d and an uncovered starting node i, a *safe path,* P, is a set of uncovered nodes in D that are parents of i (or descendants of parents of i), s.t. P is the minimum set of nodes that need to have code generated for them to make the set $d \cup \{i\}$ a safe position.

Safe paths are important because the algorithm we propose will always follow a safe path (or several noninterfering safe paths) in the process of code generation.

4.1 The Reorganization Algorithm

The reorganization algorithm is a constraint algorithm; since it is nonoptimal, some heuristic choices are incorporated. The general structure of the algorithm is to find the acceptable set of next instructions and then to heuristically choose from among them.

Assume $D_1, D_2, ..., D_n$ are parallel dags, with covered sets $C_1, ..., C_n$ and fully covered sets $F_1, ..., F_n$. The steps of the algorithm are as follows:

1. Compute the set of legal next choice nodes: N, s.t.:

   a. if $i \in D_j - C_j$ and the set of children of $i \subseteq C_j$ and $\forall i'$ (WillInterlock(i',i,n) = false), where n is the number of instructions generated since i' was emitted, then put i in N.

   b. Starting with i, find the minimum safe path from C_j. Call this Safepath(i,j).

   c. If the safe paths of two starting nodes overlap, eliminate one of the nodes generating the safe paths from N (usually the node with the longer saf path should be eliminated).

   d. If $k \in N$ is a load or store, check that it does not access a memory location out of order; if it does, remove k from N.

2. If N is empty, emit a no-op. Otherwise, choose an instruction from N. If in step 1.c a safe path has been eliminated, the next instruction should be chosen from the favored safe path.

Lemma 15: If the reorganization algorithm follows safe paths it never blocks.

Proof: By induction on the length of the code sequence emitted so far.

Basis: length = 0. Since no instructions have been emitted yet, there can be no interdependencies and no instruction choice will block.

Induction step: Assume for length = n. It is sufficient to consider the case where only two nodes remain and these nodes block (since otherwise a nonblocking node could be chosen). For two nodes to block their destinations must be sources for the other node (as in Figure 4). Thus, all we need demonstrate is that the reorganizer will never reach such a state. This is proved by contradiction of the induction hypothesis.

Assume the reorganizer reached a blocking state for nodes n_1 and n_2 with destinations d_1 and d_2 and children s_1 and s_2 respectively. (The destinations of s_1 and s_2 are d_2 and d_1 respectively). Assume that s_1 was chosen before node s_2, and consider the algorithm at the point s_2 is chosen.

To choose s_2 over n_1, s_2 must be the start of a safe path. The only safe position once s_1 and s_2 are covered is the entire dag. But there is no safe path starting with s_2 that covers the entire dag (since safe paths must involve only parents of s_2).

Since the algorithm followed safe paths for the first n nodes, node s_2 could not have been chosen over node n_1, and the lemma must hold. ∎

Thcorcm 16: The reorganization algorithm correctly emits code that avoids blocking and computcs the same function as thc original code scqucnce.

Proof: If the initial register allocation is correct, traversing the code in a bottom-up fashion will always yield a lcgal computation, provided thc reorganizer does not block, the interlocks are preserved, and aliasing effects arc avoided. Aliasing effects are avoided by checks on loads and stores (step 1d); step la checks for interlocks; blocking can not occur by Lemma 15. The bottom up traversal is correct, since code is not emitted for a node unless its children are covcrcd.

Since the algorithm cannot block, if WillInterlock is false for some amount of instruction separation, then thc algorithm must complete thc code sequcnce. Furthermore, since the code is correct, we know that a complete and correct code sequence must bc emitted. ∎

5 Implementation

We have implemented a compiling system and reorganizer for MIPS (Microprocessor without Interlocked Pipe Stages) [6], an ongoing, experimental VLSI processor project. Currently, compilers for Pascal, Fortran, and C exist. These compilers generate machine-language level instructions that ignorc the possibility of interlocks. The reorganizer is an implementation of the techniques described above; it also provides several othcr functions, such as limited instruction collapsing and instruction assembly. Although the pipeline interlocks in MIPS are straightforward, they significantly affect code generated from a standard compiler.

5.1 MIPS interlocks

MIPS has a six-stage pipcline with three active instructions occupying every other pipestage. For the purposes of this paper it is sufficient to consider only the destination-source interlocks that occur when registers arc written and then used as sources on the next instruction. Results from arithmetic operations can be written in pipestages 3 and 6, and registers can bc loaded from memory during pipestage 5. Registers are used as sources for address calculations and arithmetic operations during pipestages 3 and 6, and as sources to be stored during pipestage 4. The interlocks that arise from this pipeline structure can be summarized as follows:

Destination Field	Source Field
(i,5)	(i+1,3)
(i,6)	(i+1,4)

In Table 1 we show some typical instructions and their use of the pipeline.

Instruction	Sources	Destinations
Load 10(R2),R1	R2 during 3	R1 during 5
Add R2,R1	R1, R2 during 3	R1 during 3
Add R2,R1	R1, R2 during 6	R1 during 6
Store R1,20(R2)	R2 during 3, R1 during 4	

Table 1: Resource usage of MIPS instructions

5.2 Effectiveness of the reorganization

The reorganizer has given us an opportunity to evaluate the effectiveness of removing pipeline interlocks by compile-time analysis. In Figure 5 we show a typical sample instruction sequence, ahd the legally padded sequence generated by added in-ops. Figure6 shows the resulting machine-level dag, and the reorganized code sequence produced by the reorganizer. The reorganized code has 2 less instructions than the code using no-ops. Since all MIPS instructions take the same amount of time to execute, and occupy the same amount of instruction space, this is a 30% improvement in execution time and instruction space.

Sample instruction sequence

```
1   Ld  1(sp), R1
2   Add  #1, R1
3   St  R1, 1(sp)
4   Ld  2(sp), R2
5   Add R1, R2, R3
6   St R3, A
```

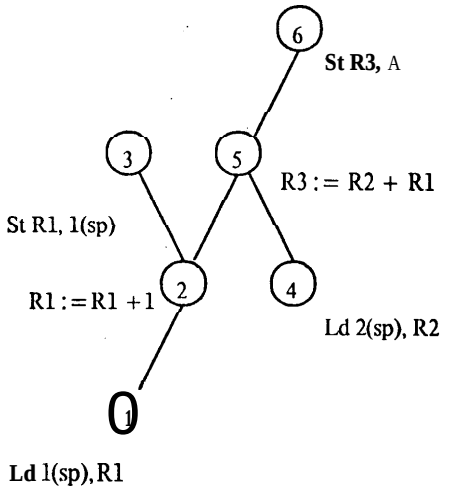
Legal instructions without reordering

```
1   Ld  1(sp), R1
2   No-op
3   Add  #1, R1
4   St  R1, 1(sp)
5   Ld 2(sp), R2
6   No-op
7   Add  R1, R2, R3
8   St R3, A
```

Figure 5: Code sequence and padded code

Table 2 is a table of empirical results for the reorganizer. The test programs consist of

1. Fibbonacci, a recursive implementation of computing a Fibonacci number,
2. Puzzle O-4, four versions of the infamous Puzzle benchmark [3] that recursively solves a cube packing problem,
3. Allo, the storage allocator from the Portable C Compiler,



Reorsanized code

```
1   Ld  1(sp), R1
2   Ld  2(sp), R2
3   Add  #1, R1
4   S t  R1, 1(sp)
5   Add  R1, R2, R3
6   St R3, A
```

Figure 6: Machine-level dag and reorganized code

4. Reader, a portion of the scanner from the Portable C Compiler.

Part of the function of our reorganizer is to pack together instruction pieces into complete' instructions; this function is similar to, but simplier than, the packing of vertical microoperations into horizontal micro-instructions. This instruction packing tends to emphasize the need for reorganization techniques because it increases the number of no-ops needed in unreorganized code. In the empirical data in Table 2, the effects of instruction packing are not considered.

6 Compiler interaction

Clearly this type of architecture and the code reorganization technique affects many parts of a compiler system including the code generator, the optimizer, and symbolic debuggers.

Program Name	Instruction count	No-ops needed without reorganization	No-ops needed with reorganization
Fibonacci	43	3	3
Puzzle 0	795	134	58
Puzzle 1	566	15	6
Puzzle 2	815	113	55
Puzzle 3	818	113	55
Al10	1272	193	86
Reader	2744	424	179

Table 2: Empirical data for reorganization

6.1 Code generation and local register allocation

In our reorganization scheme, we assumed the output of the code generators to be fixed. An important issue is the interdependency of the reorganization phase with preceding phases, i.e. if register allocation were done at this level, what could be gained from such a strategy? Different register allocation schemes can clearly influence the amount of reordering.

Consider the instructions which are generated for the statement: x [i] = k + x [j]. Our first adoption of the portable C compiler [8] produced the instruction sequence shown in Figure 7; it reuses registers as early as possible. Figure 7 also shows the reorganized code that requires three no-ops to prevent destination-source interlocks; a reorganization is not possible.

```
Code generated
Ld J, RO
Ld X(RO), R1
Ld K, RO
Add RO, R1
Ld I, RO
St R1, X(RO)
```

```
Legal instructions
Ld J, RO
No-Op
Ld X(RO), R1
Ld K, RO
No-Op
Add RO, R1
Ld I. RO
No-Op
S t R1, X(RO)
```

Figure 7: Expression Evaluation: Old Rcgistcr Assignment

A different approach is postpone the recycling of the registers and make use of more of the registers. This can be done by using different register groups (odd/even) or by cycling through the registers which are available for the evaluation of expressions. This second approach has been taken in our adoption of the code generator, and the resulting code sequence is shown in Figure 8. Two of the no-ops can be removed by reordering the code, only one no-op remains. More sophisticated register allocation algorithms often do not

Code aenerated	Legal instructions
Ld J, RO	Ld **J,** RO
Ld X(RO), **R1**	Ld K, **R2**
Ld K, R2	Ld X(RO), **R1**
Add R2, **R1**	Ld I, R3
Ld I, R3	Add R2, **R1**
St Rl, X(R3)	No-Op
	St **R1,** X(R3)

Figure 8: Expression Evaluation: New Register Assignment

dedicate registers, but use only the minimum number required for expression evaluation and leave other registers free for other purposes. Such an allocation scheme requires more careful integration of potential effects of an allocation scheme on the reorganization process.

The presence of pipeline interlocks, either in hardware of software, may change the accuracy of the usual metrics employed to choose between two alternative instruction streams. In particular, loads will tend require longer pipeline delays than instructions whose operands are in registers. For example, ignoring the effect of pipeline interlocks, loads and register-register operations may not differ in terms of size or execution time. However, when the interlocks are considered the use of load instructions tends to produce inferior code quality.

6.2 Global register allocation

Global register allocation also affects the reorganization process. When registers are globally allocated, a register may be active at the first instruction of a basic block. this occurs when the register is a destination near the end of a previous basic block. In the general case of long interlock periods, the reorganizer will have to know or find the predecessor blocks to determine what registers are affected by interlocks at the beginning of the basic block.

In most practical instances this will not 'necessary. When a basic block ends in a jump (or other nonsequential control transfer), the time to process the change of the PC will nearly always exceed the length of the longest register interlocks. Thus, the reorganizer will only need to consider interlocks that arise from sequential control flow into a basic block. These interlocks are easily computed when the previous basic block is processed.

6.3 Debugging

The code reorganization process is like an optimization in that the in tcrmediate stages of the computation are altered. Thus, the same problems that arise when trying to debug optimized code occur, and similar solutions are appropriate.

The major affect of code reorganization, on the debugging process, is to move stores with respect to other stores and computations that may fail (for example, from an overflow). These situations correspond to the type of reordering that can occur when generating code from a dag. This problem and potential solutions to the problem are discussed in depth in [7].

7 Conclusion

Modern advances in processor architecture and the constraints of VLSI design are creating new requirements for compiler and code generation techniques. A unified design approach to computer architecture allows the compiler designer to significantly influence the architecture and make hardware/software tradeoffs. The resulting architecture may then require more powerful compilers and optimizers to perform effectively.

This paper examines the process of code reorganization for a pipelined processor without pipeline interlocks. We define the problem, and propose a solution based on a postpass reorganization technique. An optimal solution to the reorganization problem is NP-complete. We give an alternative heuristic algorithm and some empirical data on its performance. Lastly, we discuss the integration of the postpass reorganizer into a compiler system.

Acknowledgments

We thank John Gill for generating the C compiler for MIPS. James Celoni, S.J. for insight on the NP-hard proof. Norman Jouppi, Jud Leonard, and the other members of the MIPS design team have provided useful . suggestions during the course of this research.

References

1. Aho, A.V. and Ullman J.D.. *Principles of Compiler Design.* Addison Wesley, Menlo Park, 1977.

2. Aho, A.V. and Johnson, J.C. "Code generation for expressions with common subexpressions." *JACM* 24, 1 (1977), 146-160.

3. Baskett, F. Puzzle: an informal compute bound benchmark. Widely circulated and run.

4. Davidson, S., Landskov, D., Shrivcr, B.D., and Mallett, P.W. "Some Experiments in Local Microcode Compaction for Horizontal Machines." *Trans. on Computers C-30,7* (July 1981), 460-477.

5. Denning, P.J. "Computer Architecture: Some Old Ideas that Haven't Quite Made It Yet." *CACM 24*, 9 (September 1981), 553-554. ACM President's Letter.

6. Hennessy, J.L., Jouppi, N., Baskett, F., and Gill, J. MIPS: A VLSI Processor Architecture. Proc. CMU Conference on VLSI Systems and Computations, October, 1981.

7. Hennessy, J.L. "Symbolic Debugging of Optimized Code." *ACM Trans. on Programming Languages and Systems* (1982). To appear.

8. Johnson, S.C. A Portable Compiler: Theory and Practice. Proc. 5th POPL Conference, ACM, January, 1978, pp. 97-104.

9. Lampson, B.W., McDaniel, G.A. and S.M. Omstein. An Instruction Fetch Unit for a High Performance Personal Computer. Tech. Rept. CSL-81-1, Xerox PARC, Jan, 1981.

10. McLellan. "IBM, A Radical Departure." *Datamation 25*, 11 (October 1979), 53-55.

11. Patterson, D.A. and Sequin C.H. RISC-I: A Reduced Instruction Set VLSI Computer. Proc. of the Eighth Annual Symposium on Computer Architecture, Minneapolis, Minn., May, 1981.

12. Patterson, D.A. and Ditzel, D.R. "The Case for the Reduced Instruction Set Computer." *Computer Architecture News 9*, 3 (October 1980).

13. Scthi, R. "Complete register allocation problems." *SIAM J. Computing 4*, 3 (1975), *226-248.*

14. Tokoro, M., Tamura, E. and Takizuka, T. "Optimization of Microprograms." *Trans. on Computers C-30*, 7 (July 1981), 491-504.

15. Widdoes, L.C. The S-l Project: Developing high performance digital computers. Proc. Compcon, IEEE, San Francisco, Feb, 1980.

16. Wulf, W.A. "Compilers and Computer Architecture." *Computer 14*, 7 (July 1981), 41-48.