Scheduling

Previously

- Value Numbering

Today: Scheduling to Minimize Register pressure

- Introduction to scheduling
- Scheduling a basic block
- Sethi-Ullman Numbering (quick review)
- Proebsting & Fischer

What makes instruction scheduling hard?

Instruction data path

IF ⇒ **Reg** ⇒ **ALU** ⇒ **DM** ⇒ **Reg2**

IF: Instruction Fetch
Reg: Instruction Decode/Register Fetch
ALU: Execute/Fetch
DM: Memory Access
Reg2: Write Back

Pipelining

Instead of:

```
Instr
        Instr
                Instr
                        Instr
```

We do:

```
Instr
    Instr
        Instr
            Instr
                Instr
```

By instruction component and cycle:

1	2	3	4	5	6	7	8	9
IF	Reg	ALU	DM	Reg				
	IF	Reg	ALU	DM	Reg			
		IF	Reg	ALU	DM	Reg		
			IF	Reg	ALU	DM	Reg	
				IF	Reg	ALU	DM	Reg

Pipelining

1	2	3	4	5	6	7	8	9
IF	Reg	ALU	DM	Reg				
	IF	Reg	ALU	DM	Reg			
		IF	Reg	ALU	DM	Reg		
			IF	Reg	ALU	DM	Reg	
				IF	Reg	ALU	DM	Reg

- Potential speed up = number of stages in the pipeline
- time to drain and fill pipeline limits speed up
- Rate through the pipeline is limited by the slowest stage
- **No individual instruction executes any faster**

Modern Processors (aside)

- Multi-issue: 2, 4, or more instructions issued per cycle
- Dynamic run-time scheduling of some window (16-64) of instructions
- Speculate branches, loads, values ...

What makes scheduling a pipeline hard?

Structural hazards: when the hardware cannot support all possible combinations of instructions in the pipeline because of resource conflicts

Data hazards: an instruction depends on the result of a previous instruction which is still in the pipeline.

Control hazards: arise due to branches and any other instruction that modifies the PC and affects which instructions should be in the pipeline.

One solution $\Longrightarrow$ Insert *bubbles*

Bubbles

Control Hazard: worst case: (btru R1, R2; inst2)

1	2	3	4	5	6	7	8	9	10
IF	Reg	ALU	DM	Reg					
	◯	◯	◯	◯					
					IF	Reg	ALU	DM	Reg

Structural Hazard
3 cycle multiply: (mul t1,r1,r2; mul t2,r3,r4)

1	2	3	4	5	6	7	8	9	10
IF	Reg	◯	◯	ALU	DM	Reg			
	◯	◯	IF	Reg	◯	◯	ALU	DM	Reg
				IF	...				

or

1	2	3	4	5	6	7	8	9	10
IF	Reg	◯	◯	ALU	DM	Reg			
	IF	Reg	◯	◯	◯	◯	ALU	DM	Reg
		IF	...						

Other solutions

Structural hazards:

- hardware functional unit replication
- compiler instruction scheduling

Data hazards:

- partial solution - hardware *forwarding*
- compiler instruction scheduling
- sensitive to accuracy of aliasing
  → runtime speculation

Control hazards:

- runtime speculation
- compiler instruction scheduling

Is there a pattern here?

Scheduling for a Pipelined Architecture

Simplification: consider only data (register and memory) hazards (*a.k.a.*, interlocks).

Goal: an efficient, compile-time algorithm for reordering instructions to minimize the number of stalls (bubbles) in the pipeline. This scheduling is performed after code generation and register allocation.

- Hennessy & Gross ('83) - $O(n^4)$ algorithm (where n is the number of instructions in the basic block). It uses lookahead to avoid deadlocking the scheduling algorithm. (Deadlock can occur because they do not represent write-write dependences in their dag).

- Gibbons & Muchnick ('86) - $O(n^2)$ worst case, $O(n)$ in practice, with nearly the same results as Hennessy & Gross.

P. B. Gibbons and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture", *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

Scheduling, Experssions, and Code Generation

How does code generation and the shape of expressions relate to scheduling and register allocation?

Code Generation

Sethi-Ullman numbering of an expression

- Determines the minimal number of registers to evaluate a tree
- Generates code using that number of registers, or *fewer*
- Assumes
  - memory can be used directly
  - no delay between a load and its use
- Works for dags by converting them into trees

Sethi-Ullman numbering

Given a binary expression tree,

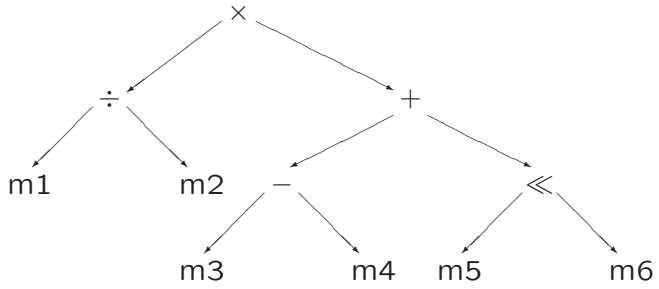
- label the left leaves 1
- label the right leaves 0
- label unary operations 1
- label the interior nodes in bottom-up order:

$$label(n) = \begin{cases} max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

- evaluate the most demanding subtree first

Example:

(m1/m2) * ((m3 -m4) + (m5 << m6))



Code Generation versus Scheduling

Sethi-Ullman is optimal on simple (unrealistic) model

- minimizes register use
- minimizes execution time

$\Longrightarrow$ assumes loads from memory are immediate

Delayed-load architectures are more complex

- issue *load*, result appears *delay* cycles later
- execution continues unless result is referenced
- premature reference causes an interlock

Many RISC systems have this property (*e.g.*, SPARC and MIPS R3000)

T.A. Proebsting and C.N. Fischer, "Linear-time, optimal code scheduling for delayed-load architectures", *in Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation"*

Scheduling

The big picture

- *interlocks* waste resources
- rearrange instructions to fill *delay* slots
- *combine* scheduling and register allocation *for expressions*
- $\Rightarrow$ *move loads as early as possible given register constraints*

Assumptions

1. input is an *expression tree* for a basic block
2. delayed-load, RISC architecture
   - register-to-register ops, *load*, & *store*
   - 1 cycle/instruction
   - non-blocking, 2 cycle *load*

So what's wrong with Sethi-Ullman?

1. modify labeling scheme for RISC (data in memory must be loaded into a register)
   - left & right leaves labeled with 1
   - interior labels as before

$$label(n) = \begin{cases} max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

2. code generation
   - more demanding subtree first
   - if $l > \mathcal{R}$, spill ($\mathcal{R}$ is # regs.)

Optimal if $delay = 0$

The problem

- *load*s will interlock if $delay > 0$

Overview of the solution

- move loads back at least *delay* slots from *ops*
- what happens to register pressure?

Brute force solution

Obvious approach

- issue all the loads
- execute all the operators

Unfortunately, this approach creates *too much* register pressure

Phase ordering

- allocate first $\Rightarrow$ poor schedule
- schedule first $\Rightarrow$ poor allocation

$\Longrightarrow$ Consider *scheduling* & *allocation together.*

Proebsting and Fischer's DLS (Delayed-Load Scheduling) Algorithm

- retain properties of Sethi-Ullman
  - contiguous evaluation
  - minimize register use
- consider two problems together[†]

The DLS algorithm

Legal ordering

- given an operator O, its children appear before O
- each load appears before the operator that uses it

The big picture

- schedule the operations (*à la* Sethi-Ullman), preserving their relative order (*ops* $\leftrightarrow$ *ops*)
- schedule the loads, preserving their relative order (*loads* $\leftrightarrow$ *loads*)
- move the loads up just enough to fill the delay slots

The DLS algorithm

The canonical order

Given $\mathcal{R}$ registers

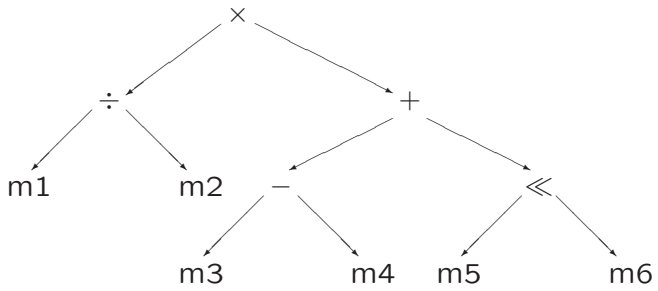
1. schedule $\mathcal{R}$ loads
2. schedule a series of (*operation; load*) pairs
3. schedule the remaining $\mathcal{R}-1$ *ops*

This keeps extra register pressure down

The algorithm

1. run Sethi-Ullman algorithm
   - calculate *minReg* for each subtree
   - create an ordering of the operators
2. put loads into canonical order
   - uses *minReg*$+1$ regs
   - requires some renaming

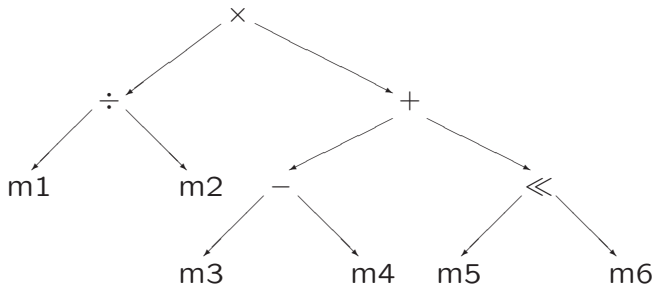
Example



Canonical ordering

	Operators		*Loads*
1.	sub	1.	load m3
2.	shift	2.	load m4
3.	add	3.	load m5
4.	div	4.	load m6
5.	mult	5.	load m1
		6.	load m2

Example



	Sethi-Ullman	DLS(3)	DLS(4)
1.	load m3, r1	load m3, r1	load m3, r1
2.	load m4, r2	load m4, r2	load m4, r2
3.	*–stall–*	load m5, r3	load m5, r3
4.	sub r1,r2,r2	sub r1,r2,r2	load m6, r4
5.	load m5, r1	load m6, r1	sub r1,r2,r2
6.	load m6, r2	*–stall–*	load m1, r1
7.	*–stall–*	shift r1,r3,r3	shift r3,r4,r4
8.	shift r1,r3,r3	load m1,r3	load m2, r3
9.	add r2,r3,r3	add r2,r1,r1	add r2,r4,r4
10.	load m1, r1	load m2,r2	div r1,r3,r3
11.	load m2, r2	*–stall–*	mult r4,r3,r3
12.	*–stall–*	div r3,r2,r2	
13.	div r1,r2,r2	mult r1,r2,r2	
14.	mult r3,r2,r2		

Optimal Spilling

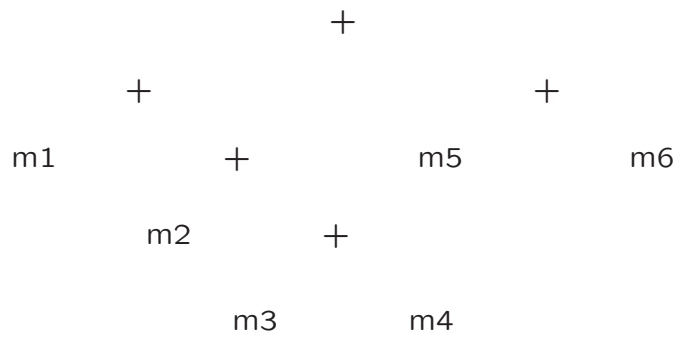
If fewer than minReg+1 registers are available

- Should we spill such that subtrees don't interlock?
- Where should the spills be introduced?
- How many interlocks must occur?
- Spilling vs. interlock - which results in the minimal number of cycles?

To minimize spill/delay costs

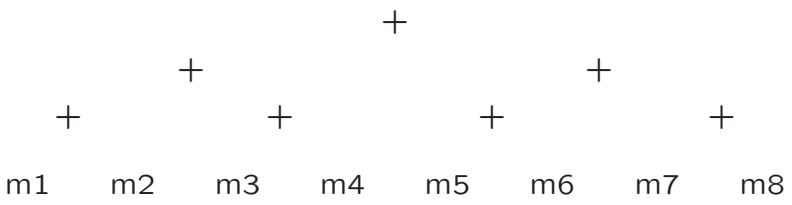
- Compute register *pressure*
  - the number of times minReg registers will be *live* during Sethi-Ullman numbering
- Schedule the tree with the minimal pressure last
- Spill the child of this node that was just evaluated if
  - minReg of this node equals the number of registers, and
  - the pressure is greater than 2 (the cost of a load and a store).
  - Otherwise if pressure is ≤ 2, take an interlock.
- Spill also if this node's minReg is greater than the number of registers, then spill the child with greatest pressure.

Register Pressure

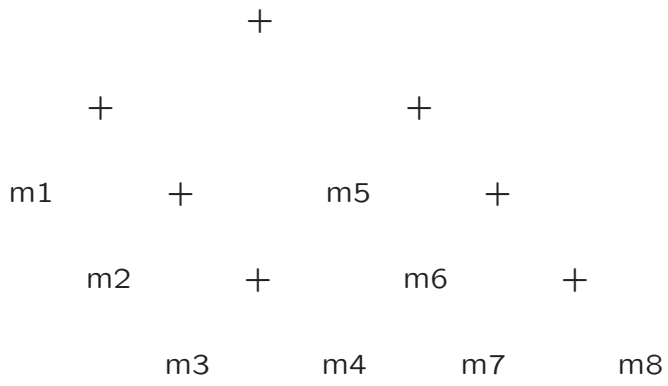


Spilling vs. Interlock Example

4 Regs	No Spills, Interlocks	Spills, No Interlocks
1.	load m1, r1	load m1, r1
2.	load m2, r2	load m2, r2
3.	load m3, r3	load m3, r3
4.	load m4, r4	load m3, r4
5.	add r1,r2,r2	add r1,r2,r2
6.	load m5, r1	load m5, r1
7.	add r3,r4,r4	add r3,r4,r4
8.	load m6, r3	load m6, r3
9.	add r2,r4,r4	add r2,r4,r4
10.	load m7, r2	load m7, r2
11.	add r1,r3,r3	store r4, tmp
12.	load m8, r1	load m8, r4
13.	*–stall–*	add r1, r3, r3
14.	add r2,r1*,r1	load tmp, r1
15.	add r3,r1,r1	add r2,r4,r4
16.	add r3,r1,r1	add r3,r4,r4
17.		add r1,r4,r4



Spilling vs. Interlock Example



Spilling vs. Interlock Example

3 Regs	No Spills, Interlocks	Spills, No Interlocks
1.	load m3, r1	load m3, r1
2.	load m4, r2	load m4, r2
3.	load m2, r3	load m2, r3
4.	add r1,r2,r2	add r1,r2,r2
5.	load m1, r1	load m1, r1
6.	add r3,r2,r2	add r3,r2,r2
7.	load m7, r3	load m7, r3
8.	add r1,r2,r2	add r1,r2,r2
9.	load m8, r1	load m8, r1
10.	–*stall*–	store r2, tmp
11.	add r3,r1∗,r1	load m6, r2
12.	load m6, r3	add r3,r1,r1
13.	–*stall*–	load m5, r3
14.	add r1,r3∗,r3	add r2,r1,r1
15.	load m5, r1	load tmp, r2
16.	–*stall*–	add r3,r1,r1
17.	add r3,r1∗,r1	add r2,r1,r1
18.	add r2,r1,r1	

Limitations

Input (*like* Sethi-Ullman)

- limited to a single basic block
  ⇒ How should it be integrated with a global allocator?
- handles *trees*, not *dags*
  ⇒ use dag-to-tree conversion, but that creates copies and recomputation

Output

- *delay* = 1 ⇒ optimal
- *delay* > 1 ⇒ optimality not guaranteed, but it is very good (>97% of optimal) on trees of 25 or fewer nodes
- non-constant *delay* causes deeper problems

General comments

- fast, simple algorithm
- clever metric for spilling when not enough registers are available
- no excuse to do worse

This work raises the bar for optimizing and non-optimizing *compilers*

Next Time

- What about intruction level parallelism (ILP) and expressions?
- Hunt, Maher, Coons, Burger, and McKinley, Optimal Huffman Tree-Height Reduction for Instruction Level Parallelism, Department of Computer Sciences Technical Report, The University of Texas at Austin, TR-08-34, 2008.