Code generation

Topics

- register allocation
  - place operands in registers
  - reduce load/store operations
- instruction scheduling
  - calculate when each instruction executes
  - enables pipelining to hide latency
  - required on VLIW architectures
- high-level languages
  - object oriented
  - functional
- code generation generators
  - automate backend construction
  - IR to code using specification

Code generation for trees

Sethi-Ullman algorithm

- generates 3-address code for expression trees
- uses minimal number of registers
- combines allocation and scheduling

Overview of algorithm

Phase 1

- compute number of registers required to evaluate a subtree without storing values to memory
- label each interior node with that number

Phase 2

- walk the tree and generate code
- evaluation order guided by labels

Sethi-Ullman Phase 1

`if n is a leaf then`
  `label(n)` $\leftarrow$ `1`

`else begin /*` *n is an interior node* `*/`
  `let` $n_1, n_2, \ldots, n_k$ `be the children of n, ordered so that`

$$\texttt{label}(n_1) \geq \texttt{label}(n_2) \geq \cdots \geq \texttt{label}(n_k)$$

$$\texttt{label(n)} \leftarrow max_{1 \leq i \leq k}(\texttt{label}(n_i) + i - 1)$$

Can compute `labels` in postorder

For $n \leq 2$, `label` is defined recursively as:

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

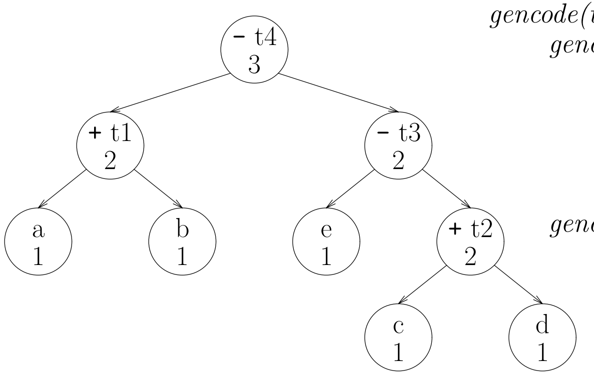
`label` $=$ *minReg* (minimum # of registers)

Sethi-Ullman Phase 2

`REG` = current register number (initialized to 1)

```
procedure gencode(n)

  if n is leaf "name"
    /* case 0 — just load it */
    emit(load, REG, name);

  else if n is interior node "op n1 n2" then

    if label(n1) ≥ label(n2) then
      /* case 1 — generate left child first */
      gencode(n1);     REG = REG+1;
      gencode(n2);     REG = REG-1;
      emit(op, REG, REG, REG+1);

    else label(n1) < label(n2) then
      /* case 2 — generate right child first */
      gencode(n2);     REG = REG+1;
      gencode(n1);     REG = REG-1;
      emit(op, REG, REG+1, REG);

    endif
  endif
```

Sethi-Ullman Example




```
gencode(t4)                         case 1
    gencode(t1)                     case 1
        gencode(a)                  case 0
            load r1, a
        gencode(b)                  case 0
            load r2, b
        add r1, r1, r2
    gencode(t3)                     case 2
        gencode(t2)                 case 1
            gencode(c)              case 0
                load r2, c
            gencode(d)              case 0
                load r3, d
            add r2, r2, r3
        gencode(e)                  case 0
            load r3, e
        sub r2, r3, r2
    sub r1, r1, r2
```


Improved code generation for trees

Delayed-load architectures

- issue `load`, result appears k cycles later
- attempt to access target of load early causes hardware to stall (*interlock*)
- k increases for modern microprocessors

Apply instruction scheduling

- move load back at least k slots from *op*
- to maintain legality, may need more registers

Naive approach

- issue all loads, then execute all operators
- will use too many registers

Phase ordering problem

- allocate registers first $\Rightarrow$ many stalls
- schedule instructions first $\Rightarrow$ many registers

Delayed load scheduling (DLS)

Approach

1. schedule the operations (*à la* Sethi-Ullman)
2. schedule the loads

Legal ordering

- children of an operator appear before it
- each load appears before operator that uses it

The final schedule

- preserves relative order of operations
  (*ops* $\leftrightarrow$ *ops*)
- preserves relative order of loads
  (*loads* $\leftrightarrow$ *loads*)
- changes relative order of loads to operations

T.A. Proebsting and C.N. Fischer, "Linear-time, optimal code scheduling for delayed-load architectures," in Proceedings of SIGPLAN PLDI'91

The DLS algorithm

The canonical order

Given $\mathcal{R}$ registers

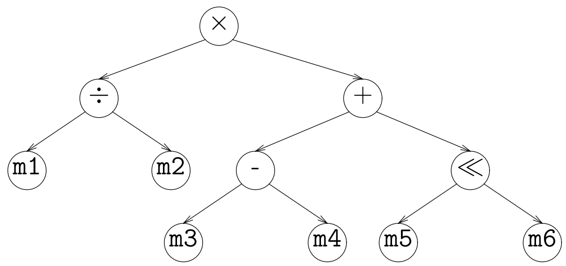
1. schedule $\mathcal{R}$ loads
2. schedule a series of (*op,load*) pairs
3. schedule the remaining $\mathcal{R} - 1$ *ops*

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

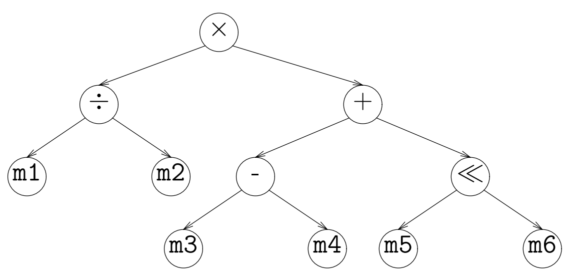
DLS example



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

DLS example



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

Limitations

Input

(*like* Sethi-Ullman)

- handles *trees*, not *dags*
- limited to a single basic block
- values not kept in registers

Output

- *delay* $> 1 \;\Rightarrow$ optimality not guaranteed
- non-constant *delay* causes deeper problems

Strengths

- fast, simple algorithm
- clever metric for spilling
- no excuse to do worse

This work raises the bar for non-optimizing compilers