

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) ← 1
else begin /* n is an interior node */
  let  $n_1, n_2, \dots, n_k$  be the children of n,
  ordered so that
  Label( $n_1$ ) ≥ Label( $n_2$ ) ≥ ... ≥ Label( $n_k$ )
  Label(n) ←  $\max_{1 \leq i \leq k} (\text{Label}(n_i) + i - 1)$ 
```

Can compute labels in postorder

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

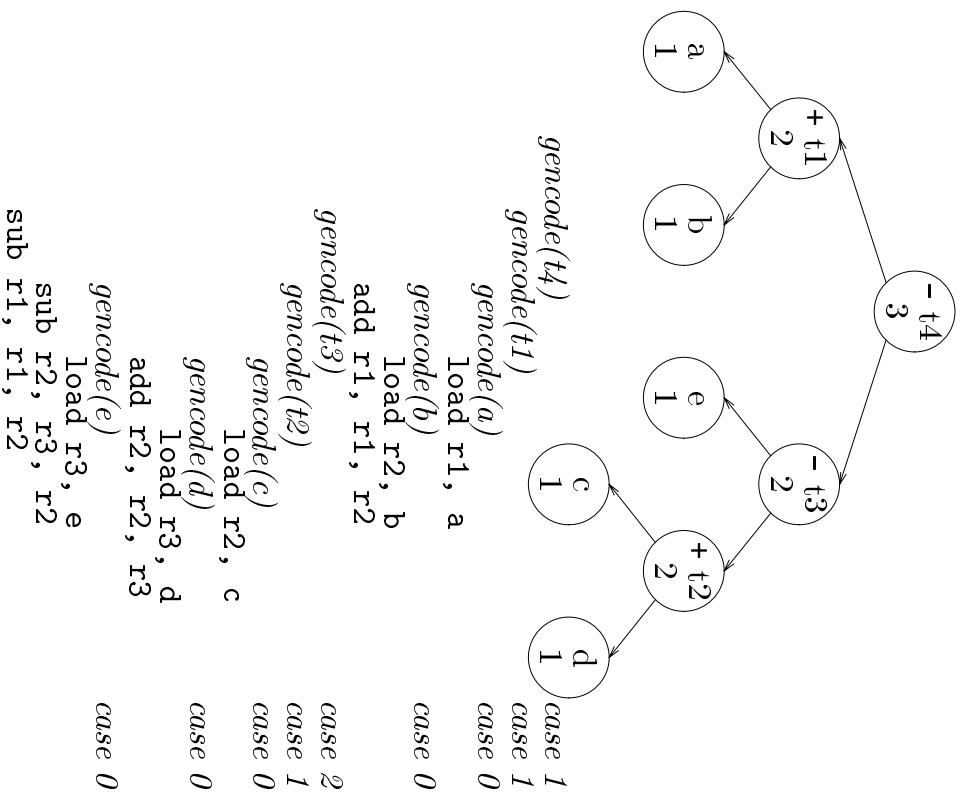
$$\text{Label}(n) = \begin{cases} l_1 + 1 & \text{if } l_1 = l_2 \\ \max(l_1, l_2) & \text{if } l_1 \neq l_2 \end{cases}$$

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  $n_1$   $n_2$ " then
    if Label( $n_1$ ) ≥ Label( $n_2$ ) then
      /* case 1 — generate left child first */
      gencode( $n_1$ )
      REG = REG+1
      gencode( $n_2$ )
      REG = REG - 1
      emit(op, REG, REG, REG+1)
    else Label( $n_1$ ) < Label( $n_2$ ) then
      /* case 2 — generate right child first */
      gencode( $n_2$ )
      REG = REG+1
      gencode( $n_1$ )
      REG = REG - 1
      emit(op, REG, REG+1, REG)
  endif
endif
```

Sethi-Ullman Example



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* ↔ *ops*)
- preserves relative order of loads (*loads* ↔ *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

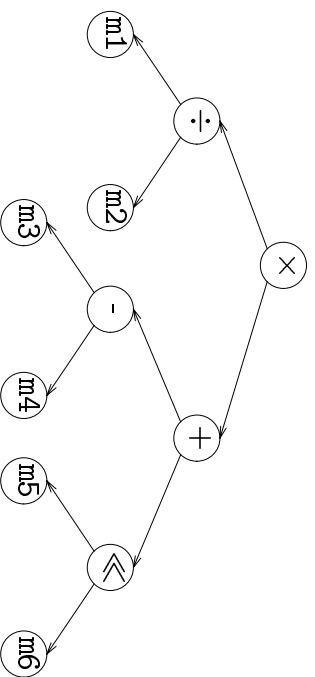
1. schedule \mathcal{R} loads
2. schedule a series of $(op, load)$ pairs
3. schedule the remaining $\mathcal{R} - 1$ *ops*

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

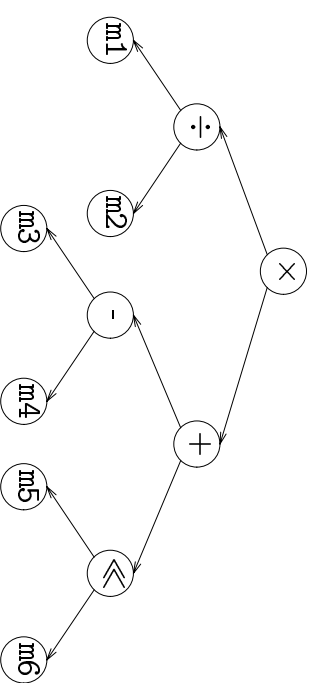
Example



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, r1, r2 | sub r1, r1, r2 | load m6, r4 |
| 5. | load m5, r2 | load m6, r2 | sub r1, r1, r2 |
| 6. | load m6, r3 | -stall- | load m1, r2 |
| 7. | -stall- | shift r2, r3, r2 | shift r3, r3, r4 |
| 8. | shift r2, r2, r3 | load m1, r3 | load m2, r4 |
| 9. | add r1, r1, r2 | add r1, r1, r2 | add r1, r1, r3 |
| 10. | load m1, r2 | load m2, r2 | div r2, r2, r3 |
| 11. | load m2, r3 | -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