Lecture 9: Improved Code Generation
A. Pnueli

Better Code Generation Requires Greater Context

Over expressions:
- Optimal ordering of subtrees
- Common subexpression elimination

Over basic blocks:
- Global register allocation, register coloring

Over procedures:
- Register tracking with last-use information
- Common subexpression elimination

Over the program:
- Inter-procedural flow analysis

Decomposition into Basic Blocks

To partition a program into basic blocks:
- Call the first instruction (quadruple) in a basic block
- Its leader
- Any instruction that follows a jump is a leader
- Any instruction that is the target of a jump is a leader
- The first instruction in the program is a leader

Over the program:
- Register tracking with last-use information
- Common subexpression elimination

Over basic blocks:
- Optimal ordering of subtrees

Over procedures:
- Global register allocation, register coloring

Over the program:
- Inter-procedural flow analysis

Next leader

In the presence of flow of control, values of variables can depend on multiple points of definition and points of use of variables.

---

Honors Compilers, NYU, Fall, 2009

Transformations on Basic Blocks

Dead-Code Elimination (within basic blocks)

Dead-Code Elimination

- Within block:
  - Computed by a single backwards pass over quadruples and symbol table.
  - Next-use information is an annotation over quadruples.

- Next-use annotation is a mark of deadness.
- Router: x, y, z were dead in previous basic blocks.
- No reference to x.
- Dead: can be removed later.

- Examples:
  - Remove quadruple: x := y + z if x is dead.

Simple Symbolic Interpretation

- Next-use of y, z is q, mark both as live.
- Value of x is dead in symbol table (previous value has no next use).
- Record next uses of x, y, z from symbol table into quadruple.

- For quadruple q: x := y op z:
  - On exit from block, all temporaries are dead (no next-use).
  - Operands next-use (later quadruple number).
  - Operands liveness (boolean).

- Information:
  - Each operand in a quadruple and symbol table carries additional
    information.
  - Use symbol table to annotate status of variables.

Honors Compilers, NYU, Fall, 2009

237

Honors Compilers, NYU, Fall, 2009

238

Honors Compilers, NYU, Fall, 2009

239

Honors Compilers, NYU, Fall, 2009

240
Improved Code Generation

Register Allocation over Basic Block:

1. Tracking

- Goal is to minimize use of registers and memory references
- Doubly linked data structure

Heuristics:
- Choose variable whose next use is farthest away
- Choose variable whose next use is in memory (spill) and use register
- Else find a register that holds a live variable, store variable in memory, use register
- Else if there is a register that holds a dead variable, use it
- Else if there is an available register, use it
- Else if the same for Z, provided architecture supports
- Else if no next use of Y, use Y
- Else if Y contains no other variable and Y is not live, and there is no next use of Y, use Y
- Else if Y is in R, R contains no other variable and Y is not live, and there

getreg:

- Procedure getreg determines optimal choice to hold result of next quadruple
- Procedure getreg determines „optimal“ choice to
- For each variable, indicate location of current value
- For each variable, indicate location of current value
- For each register, indicate current contents (set of variables with equal values)
- For each register, indicate current contents (set of variables with equal values)
- For each register, indicate current contents (set of variables with equal values)
- Store registers that contain live values which have no copy in memory

Using getreg:

- Call getreg to obtain target register
- For x := y op z:
  - Find current location of y, generate load into register if in memory
  - Ditto for z, except that should use register other than R
  - Update address descriptor for y, to indicate it resides in R
  - Dito for z, except that should use register other than R
  - For x := y op z:
  - Emit instruction
  - Update register descriptor for R, to indicate it holds x
  - Update address descriptor for R, to indicate it resides in R
  - Update register descriptor for R, to indicate it holds x
  - Update address descriptor for R, to indicate it resides in R
  - Update register descriptor for R, to indicate it holds x
  - Update address descriptor for R, to indicate it resides in R

<table>
<thead>
<tr>
<th>Register</th>
<th>Generated Code</th>
<th>Address Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>MOVa, R0</td>
<td>R0 contains a</td>
</tr>
<tr>
<td>R1</td>
<td>MOVa, R1</td>
<td>R1 contains a</td>
</tr>
<tr>
<td>R0</td>
<td>MOVa, R0</td>
<td>R0 contains a</td>
</tr>
<tr>
<td>R1</td>
<td>MOVa, R1</td>
<td>R1 contains a</td>
</tr>
<tr>
<td>R0</td>
<td>MOVa, R0</td>
<td>R0 contains a</td>
</tr>
<tr>
<td>R1</td>
<td>MOVa, R1</td>
<td>R1 contains a</td>
</tr>
</tbody>
</table>
DAG Construction

For Body of Loop

- DAG for Body of Loop

Example: dot product

\[
\text{prod} := 0; \quad \text{for } j := 1 \ldots 20 \text{ loop }
\]
\[
\text{prod} := \text{prod} + a(j) \times b(j);
\]
\[
\text{endloop};
\]

Quadruples:

1. prod := 0; - basic block leader
2. j := 1;
3. start : T1 := 4 * j;
4. T2 := a(T1);
5. T3 := T2 + T1;
6. T4 := b(T1);
7. T5 := T4 + T3;
8. T6 := prod + T5;
9. prod := T6;
10. T7 := j + 1;
11. j := T7;
12. if \( j \leq 20 \) goto start

Computing Dependencies in a Basic Block:

- DAG for Body of Loop
- For x := y
- Add x to list of labels of node which currently holds y
- For x := y
- Add x to list of labels for new node
descendants y (need hash scheme)
- Create new node for "op" or find an existing one with
node labeled z
- Find node labeled z or create one

Common subexpressions identified

The DAG

Intermediate nodes are labeled with operators and
operands.
Leaves are labeled with identifiers and constants.

Basic block optimization
Intermediate code optimization:
- Use directed acyclic graph (dag) to recognize common
subexpressions and remove redundant quadruples.
- Basic block is labeled with identifiers.
- DAG construction for body of loop

For Body of Loop

Forward pass over basic block
Add x to list of labels of node which currently holds y
For x := y
Add x to list of labels for new node descendants y (need hash scheme)
Create new node for "op" or find an existing one with
node labeled z
Find node labeled z or create one
Find node labeled y or create one
For x := y op z:
Forward pass over basic block

Use directed acyclic graph (dag) to recognize common
subexpressions and remove redundant quadruples.

The DAG

DAG Construction

Computing Dependencies in a Basic Block:

- DAG for Body of Loop
- For x := y
- Add x to list of labels of node which currently holds y
- For x := y
- Add x to list of labels for new node
descendants y (need hash scheme)
- Create new node for "op" or find an existing one with
node labeled z
- Find node labeled z or create one

Common subexpressions identified

The DAG
Register assignment is equivalent to graph coloring.

Each variable is a node in the graph.

There is an edge between two nodes if the lifetime of the variables overlap.

Lifetimes overlap.

Two variables cannot be assigned the same register if their lifetimes overlap.

To reuse registers, need to know lifetime of variables (set of instructions in program).

Optimal use of registers in subprogram: keep all variables in

Coloring

Using Global Information: Register Coloring

W is often a power of 2 (deephole optimization).

An array of 1 + 1 + T;

The following requires 19 quadruples.

For k in los..hig loop

end loop

A(k): = B(k) + T;

A. Programmers don’t produce common subexpressions, code generators do!

A, B, C, D arrays (i = 0..n, k = 0..m).

A loop invariant is an expression that is always true for the

loop body.

A. Programmers don’t produce common subexpressions, code generators do!

A loop invariant is an expression that is always true for the

loop body.

A. Programmers don’t produce common subexpressions, code generators do!

A loop invariant is an expression that is always true for the

loop body.

A. Programmers don’t produce common subexpressions, code generators do!

A loop invariant is an expression that is always true for the

loop body.
Graph Coloring

Given a graph and a set of $N$ colors, assign a color to each vertex so that not two connected vertices are colored by the same color.

Problem is $NP$-Complete

Fast heuristics algorithm (Chaitin) is usually linear:

Better Approach to Spilling

1. Use loop structure to estimate usage.
2. Spill variables with lowest usage count.
3. Need to place $R - N$ variables in memory.
4. Compute required number of colors in second pass: $R$.

Example

Order of removal: $B, C, A, E, F, D$

Order of removal: $C, A, E, F, D$

D (no constraint) $D$ E (D) A, F (D) E (D) A)
C (D, A) B, A, E, F (D)

Assume $3$ colors are available: assign colors in reverse order, constrained by already colored nodes.

$D$ (no constraint) $F$ (D) E (D) A, F (D) E (D) A)
C (D, A) B, A, E, F (D)

Better Approach to Spilling

If at any point a node has more than $N - 1$ neighbors, need to be deleted from graph. Start with node with smallest number of neighbors.

Fast heuristics algorithm (Chaitin) is usually linear:

Better Approach to Spilling

If at any point a node has more than $N - 1$ neighbors, need to be deleted from graph. Start with node with smallest number of neighbors.

Fast heuristics algorithm (Chaitin) is usually linear:

Better Approach to Spilling

If at any point a node has more than $N - 1$ neighbors, need to be deleted from graph. Start with node with smallest number of neighbors.