Improving Code Generation
Better code generation requires greater context

- Interprocedural flow analysis
- Over the program:
  - Global register allocation, register coloring
- Over procedures:
  - Register tracking with last-use information
  - Common subexpression elimination
- Over basic blocks:
  - Optimal ordering of subtrees
- Over expressions:

A. Pnueli
Better code generation requires information about multiple points of definition and points of use of variables.

In the presence of flow of control, values of variables can depend on multiple points in the program.

A basic block is a single-entry, single-exit code fragment. Values computed within a basic block have a single origin. More constant folding and common subexpression elimination.

Basic Blocks

A basic block is a single-entry, single-exit code fragment. Values computed within a basic block have a single origin. More constant folding and common subexpression elimination. Better register use.

Better code generation requires information about points of definition and points of use of variables.

A basic block is a single-entry, single-exit code fragment. Values computed within a basic block have a single origin. More constant folding and common subexpression elimination. Better register use.

x := y * 2;

label1:

x := y * 2;

- Here x = 24

- Can't tell, y may be different

- 24?

x := 12;

y := 12;

Better code generation requires information about points of definition and points of use of variables.
Decomposition into Basic Blocks

To partition a program into basic blocks:

Call the first instruction (quadruple in a basic block)

The first instruction in the program is a leader

Any instruction that is the target of a jump is a leader

Any instruction that follows a jump is a leader

Any instruction following a call to a procedure with side effects is a leader

Any instruction that follows, up to but not including the next leader

Instructions that follow, up to but not including the next leader
Lecture 9: Improving Code Generation

Transformations on Basic Blocks

A. Pnueli

Honors Compilers, NYU, Spring, 2007

Transformations on Basic Blocks

- Common subexpression elimination: recognize redundant re-computations. Replace with a single redundant expression elimination.

- Dead-code elimination: recognize computations whose results are never used. Remove associated quadruples.

- Interchanging statements, for better scheduling.

- Renaming of temporaries, for better register usage.

- All of the above require symbolic execution of the basic block to obtain definition/use information.

- All of the above require symbolic execution of the basic block to obtain definition/use information.
Simple Symbolic Interpretation: next-use Information

- If $x$ is computed in quadruple $i$, and is an operand of quadruple $j > i$, its value must be preserved (register or memory) until $j$.
- If $x$ is computed at $k > i$, the value computed at $i$ has no further use, and can be discarded (i.e., register reused).

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

- next-use of \( y, z \) is \( q \) if both are live.
- Mark \( x \) as dead in symbol table (previous value has no next-use).
- Mark all temporaries as dead (no next-use).
- For quadruple \( q: x\! :=\! y\! op\! z \):
  - Record next uses of \( x, y, z \) from symbol table into quadruple.
  - Mark \( x \) as dead in symbol table (previous value has no next-use).
  - next-use of \( y, z \) is \( q \), mark both live.
- On exit from block, all temporaries are dead (no next-use).
- Operand next-use (later quadruple number)
- Operand liveness (boolean)
- Information:
  - Each operand in a quadruple and symbol table carries additional information.
  - Use symbol table to annotate status of variables.
Improving Code Generation

Dead-Code Elimination (within basic blocks)

After elimination, needs to update/recompute next-use information.

Examples:

- If \( x \) is a temporary, not referenced in any later quadruple, not referenced in any later quadruple.

\[
\begin{align*}
x & : y + 1; \\
x & : y + 1; \\
x & : z + 2 \\
\end{align*}
\]

- \( x \) is dead. can be removed.
Goal is to minimize use of registers and memory referencing references.

Doubly linked data structure:

For each register, indicate current contents (set of variables with equal values): register descriptor.

For each variable, indicate location of current value: memory and/or registers: address descriptor.

Procedure getreg determines “optimal” choice to hold result of next quadruple.

Register Allocation over Basic Block:

A. Pnueli

Lecture 9: Improving Code Generation

Honors Compilers, NYU, Spring, 2007
Choose variable whose next use is farthest away. •

memory (spill), and use register.

Else find a register that holds a live variable, store variable in memory, use $R_i$. •

If $y$ is in $R_i$, $R_i$ contains no other variable, and $y$ is also in

Else, if there is a register $R_k$ that holds a dead variable, use it. •

Else, if there is an available register $R_j$ use it.

Else, try the same for $z$, provided architecture supports

operation. •

Else, if no next use of $y$, use $R_i$. •

If $y$ is in $R_i$, $R_i$ contains no other variable, $y$ is not live, and there

For quadruple $x := y \ op z$, •

Getreg: Heuristics

A. Pnueli
Lecture 9: Improving Code Generation

A. Pnueli

On block exit, store registers that contain live values.

- For $x := y$, emit instruction to obtain target register $R$
  - Call `getreg` to obtain target register $R$
  - Find current location of $y$, generate load into register $R$
  - Update register descriptor for $R$, to indicate it holds $x$.
  - Update address descriptor for $R$ to indicate it resides in $R$
  - Update address descriptor for $x$ to indicate it resides in $R$

- For $x := z$
  - Ditto for $z$, except that should use register other than $R$
  - Ditto for $z$, except that should use register other than $R$

Using 

- On block exit, store registers that contain live values.
  - Call `getreg` to obtain target register $R$
  - Find current location of $y$, generate load into register $R$
  - Update register descriptor for $R$, to indicate it holds $x$.
  - Update address descriptor for $R$ to indicate it resides in $R$.
  - Update address descriptor for $x$ to indicate it resides in $R$.
The assignment \[ d := (a - b) + (a - c) \] can be translated to

\[
\begin{align*}
    t &:= a - b; \\
u &:= a - c; \\
v &:= t + u; \\
d &:= v + u;
\end{align*}
\]

### Using getreg

<table>
<thead>
<tr>
<th>Statement</th>
<th>Code Generated</th>
<th>Address Descriptor</th>
<th>Register Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t := a - b )</td>
<td>MOV a, R0</td>
<td>t in R0</td>
<td>Registers empty</td>
</tr>
<tr>
<td>( u := a - c )</td>
<td>SUB b, R0</td>
<td>t in R0</td>
<td>R0 contains t</td>
</tr>
<tr>
<td>( v := t + u )</td>
<td>MOV a, R1</td>
<td>R0 contains t</td>
<td>R0 contains u</td>
</tr>
<tr>
<td>( d := v + u )</td>
<td>ADD c, R1</td>
<td>R0 contains v</td>
<td>R1 contains u</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
    t &= a - b; \\
u &= a - c; \\
v &= t + u; \\
d &= v + u;
\end{align*}
\]
Computing Dependencies in a Basic Block:

The DAG

Identifiers:
- Intermediate nodes are labeled with operators and internal nodes are labeled with identifiers and constants.
- Leaves are labeled with identifiers and constants.

Improved block $\leftrightarrow$ Improved block (dag) $\leftrightarrow$ Improved block (assembly)

Intermediate code optimization:
- Use directed acyclic graph (dag) to recognize common subexpressions and remove redundant quadruples.

A. Pnueli
A. Pnueli

DAG Construction

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

FOR x := y, z:

- Find node labeled y, or create one

- Find node labeled z, or create one

- Create new node for “op”, or find an existing one with descendants y, z (need hash scheme)

FOR x := y, z:

- Forward pass over basic block

FOR x := y, z:

- Find node labeled y, or create one

FOR x := y:

- Find node labeled x, or create one

FOR x := y:

- Add x to list of labels for new node
Example: dot product

prod := 0;
for j in 1...20 loop
  prod := prod + a(j) * b(j);
endloop;

- assume 4-byte integer
• Common subexpressions identified

DAG for Body of Loop
Any topological sort of the DAG is a legal evaluation order.

A node without a label is a dead value.

Prefer the label of a live variable over a temporary.

Fewer quadruples, fewer temporaries.
Programmers don't produce common subexpressions, code generators do!

A, B: array(lo1..hi1, lo2..hi2)

\[
A[B(i,j)] = \text{base-a} + (i - \text{lo1}) * (hi2 - \text{lo2} + 1) + j - \text{lo2}
\]

• A[B(i,j)] is at location:

\[
\text{base-a} + (i - \text{lo1}) * (hi2 - \text{lo2} + 1) + j - \text{lo2}
\]

• w is often a power of 2 (peephole optimization)

• w is a loop invariant (loop optimization)

• Can reduce to 1 with a DAG.

\[
\text{base-a} + (i - \text{lo1}) * (hi2 - \text{lo2} + 1) + j - \text{lo2}
\]

The following requires 19 quadruples:

\[
\text{for } k \text{ in } \text{lo2..hi2} \text{ loop}
\]

\[
A[i,k] := B[i,k] + 1;
\]

end loop;

w is often a power of 2 (peephole optimization).
Beyond Basic Blocks: Data Flow Analysis

Requires complex data structures and algorithms

- Loop invariant computations
- Live-dead analysis
- Common subexpressions elimination
- Constant folding
- Algorithms on graphs

Can compute global properties of programs as iterative

Basic blocks are nodes in the flow graph

A. Pnueli

Lecture 9: Improving Code Generation
Register assignment is equivalent to graph coloring.

Corresponding variables overlap.

- There is an edge between two nodes if the lifetime of the variables overlap.
- Each variable is a node in the graph.
- Lifetime information is translated into interference graph.

Register assignment is equivalent to graph coloring.

Lifetime overlap.

- Two variables cannot be assigned the same register if their lifetimes overlap.
- To reuse registers, need to know lifetime of variables.
- Optimal use of registers in subprograms.
- Keep all variables in registers throughout program.

Using Global Information: Register Coloring.
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 heuristic algorithm (Chaitin) is usually linear:

- If at any point a node has more than $N - 1$ neighbors, need to free a register (spill). Can then remove node and continue.
- Iterate until graph is empty, then assign colors in inverse order of neighbors.
- Any node with fewer than $N - 1$ neighbors is colorable, so can be deleted from graph. Start with node with smallest number of neighbors.
- If at any point a node has more than $N - 1$ neighbors, need to free a register (spill). Can then remove node and continue.

Pnueli

Honors Compilers, NYU, Spring, 2007
Order of removal: B, C, A, E, F, D

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

Example
Use loop structure to estimate usage:

- Spill variables with lowest usage count.
- Need to place \( N - R \) variables in memory

Compute required number of colors in second pass:

Better Approach to Spilling

A. Pnueli