Improving Code Generation

Honors Compilers
April 16\textsuperscript{th} 2002
Better code generation requires greater context

Over expressions:
- optimal ordering of subtrees

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

Over procedures:
- global register allocation, register coloring

Over the program:
- Interprocedural flow analysis
Basic blocks

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

In the presence of flow of control, value of a variable can depend on multiple points in the program:

\[
y := 12;
x := y * 2; \quad \text{-- here } x = 24
\]

\[
\text{label1: ...}
\]

\[
x := y * 2; \quad \text{-- 24? Can’t tell, } y \text{ may be different}
\]

A basic block is a single-entry code fragment: values that are computed within a basic block have a single origin: more constant folding and common subexpression elimination, better register use. Has either one exit or exit is a jump which may have two targets (conditional jump).
Finding basic blocks

To partition a program into basic blocks:

- Call the first instruction (quadruple) in a basic block its leader
- 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
- In the presence of procedures with side-effects, every procedure call ends a basic block
- A basic block includes the leader and all instructions that follow, up to but not including the next leader
Transformations on basic blocks

Common subexpression elimination: recognize redundant computations, replace with single temporary
Dead-code elimination: recognize computations not used subsequently, remove quadruples
Interchange 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
Simple symbolic interpretation: next-use information

If x is computed in quadruple $i$, and is an operand of quadruple $j, j > i$, its value must be preserved (register or memory) until $j$.

If x is computed at $k, k > i$, the value computed at $k$ has no further use, and be discarded (i.e. register reused)

Next-use information is annotation over quadruple and symbol table.

Computed on one backwards pass over quadruple
Computing next-use

Use symbol table to annotate status of variables
Each operand in a quadruple carries additional information:
- Operand liveness (boolean)
- Operand next use (later quadruple)

On exit from block, all temporaries are dead (no next-use)

For quadruple $q: x := y \text{ op } z$;
- Record next uses of $x, y, z$ into quadruple
- Mark $x$ dead (previous value has no next use)
- Next use of $y$ is $q$; next use of $z$ is $q$; $y, z$ are live
Register allocation over basic block: tracking

Goal is to minimize use of registers and memory references

Doubly linked data structure:

- For each register, indicate current contents (set of variables): 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
Getreg: heuristics

For quadruple $x := y \, op \, z$:

- if $y$ is in $R_i$, $R_i$ contains no other variable, $y$ is not live, and there is no next use of $y$, use $R_i$
- Else if there is an available register $R_j$, use it
- Else if there is a register $R_k$ that holds a dead variable, use it
- If $y$ is in $R_i$, $R_i$ contains no other variable, and $y$ also in memory, use $R_i$.
- Else find a register that holds a live variable, store variable in memory (spill), and use register
  - Choose variable whose next use is farthest away
Using getreg:

For \( x := y \text{ op } z; \)
- Call getreg to obtain target register \( R \)
- Find current location of \( y \), generate load into register if in memory, update address descriptor for \( y \)
- Ditto for \( z \)
- Emit instruction
- Update register descriptor for \( R \), to indicate it holds \( x \)
- Update address descriptor for \( x \) to indicate it resides in \( R \)

For \( x := y; \)
- Single load, register descriptor indicates that both \( x \) and \( y \) are in \( R \).

On block exit, store registers that contain live values.
Computing dependencies in a basic block: the dag

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

Intermediate code optimization:

- basic block => dag => improved block => assembly

Leaves are labeled with identifiers and constants.

Internal nodes are labeled with operators and identifiers.
Dag construction

Forward pass over basic block

For \( x := y \text{ op } z; \)
- Find node labeled \( y \), or create one
- Find node labeled \( z \), or create one
- Create new node for \( \text{op} \), or find an existing one with descendants \( y, z \) (need hash scheme)
- Add \( x \) to list of labels for new node
- Remove label \( x \) from node on which it appeared

For \( x := y; \)
- Add \( x \) to list of labels of node which currently holds
Example: dot product

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

-- assume 4-byte integer

Quadruples:
prod := 0; -- basic block leader
J := 1;
start: T1 := 4 * j; -- basic block leader
T2 := a(T1);
T3 := 4 * j; -- redundant
T4 := b(T3);
T5 := T2 * T4;
T6 := prod + T5
prod := T6;
T7 := j + 1;
j := T7
If j <= 20 goto start:
Dag for body of loop

Common subexpression identified
From dag to improved block

Any topological sort of the dag is a legal evaluation order.

A node without a label is a dead value.

Choose the label of a live variable over a temporary.

\[
\text{start: } \begin{align*}
T1 &:= 4 \times j; \\
T2 &:= a [ T1] \\
T4 &:= b [ T1] \\
T5 &:= T2 \times T4 \\
\text{prod} &:= \text{prod} + T5 \\
J &:= J = 1 \\
\text{If } j \leq 20 \text{ goto start: }
\end{align*}
\]

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

A, B : matrix (lo1 .. hi1, lo2 .. hi2); -- component size in bytes
A (j, k) is at location:

- \text{base_a} + ((j - lo1) * (hi2 - lo2 + 1) + k - lo2) * w

The following requires 19 quadruples:

\begin{verbatim}
for k in lo .. hi loop
    A (j, k) := 1 + B (j, k);
end loop;
\end{verbatim}

Can reduce to 11 with a dag

base_a + (j - lo1) * (hi2 - lo2 +1) * w is loop invariant (loop optimization)

w is often a power of two (peephole optimization)
Beyond basic blocks: data flow analysis

Basic blocks are nodes in the flow graph.
Can compute global properties of program as iterative algorithms on graph:
  - Constant folding
  - Common subexpression elimination
  - Live-dead analysis
  - Loop invariant computations

Requires complex data structures and algorithms
Using global information: register coloring

Optimal use of registers in subprogram: keep all variables in registers throughout.

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

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

Lifetime information is translated into interference graph:

- Each variable is a node in a graph.
- There is an edge between two nodes if the lifetimes of the corresponding variables overlap.

Register assignment is equivalent to graph coloring.
Graph coloring

Given a graph and a set of N colors, assign a color to each vertex so two vertices connected by an edge have different colors.

Problem is NP-complete

Fast heuristic algorithm (Chaitin) is usually linear:

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

Order of removal: B, C, 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, E) C (D, A ) B (C)
Better approach to spilling

Compute required number of colors in second pass: \( R \)

Need to place \( R - N \) variables in memory

Spill variables with lowest usage count.

Use loop structure to estimate usage.