Improving Code Generation

A basic block is a single-entry, single-exit code fragment. Values computed within a basic block have a single origin. More consistent folding and common subexpression elimination require information about points of definition and points of use of variables.

Interprocedural flow analysis

Over the program:

- Global register allocation, register coloring
- Procedure-based register tracking with last-use information

Over procedures:

- Register tracking with last-use information
- Common subexpression elimination

Over basic blocks:

- Optimal ordering of subtrees
- Over expressions:

Better Code Generation Requires Greater Context
After elimination, needs to update/recompute `next-use` information.

No reference to `x` — `Dead`, can be removed later

Examples:

Remove quadruple `y := z` if `x` is dead.

### Simple Symbolic Interpretation

Within block,

- Compiled by a single backwards pass over quadruples and symbol table.

- `next-use` information is annotated over quadruples.

Within block:

- Used (`x` is computed at `k` and has no later reference to `x`).

- Not used (`x` is computed at `k` but has no later reference to `x`).

- Interchange statements for better scheduling

- Renaming of temporaries, for better register usage

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

- Re-computation elimination: Recognize redundant computation

- Common subexpression elimination: Recognize redundant re-computation

- Symbolic `next-use` information

- Used in quadruple `g: x := y op z`:

- Mark `x` as dead in symbol table (previous value has no next use)

- Record `next` uses of `x`, `y` from symbol table into quadruple

- For quadruple `g: x := y op z`:

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

- Operands next-use (later quadruple number)

- Operands liveness (boolean)

Each operand in a quadruple and symbol table carries additional information:

- Use symbol table to annotate status of variables

### Dead-Code Elimination (within basic blocks)

Remove quadruple `x := y op z` if `x` is `dead`.

Examples:

- If `x` is a temporary, not referenced in any later quadruple.
- `x := y + 1` — Dead. Can be removed.
- `x := z + 1` — Dead. Can be removed.
- `y := x + 1` — Quadruple.
- `x := y` — Quadruple.
- `x := z` — Quadruple.

### Computing next-use information

- Uses symbol table to annotate status of variables

- Each operand in a quadruple and symbol table carries additional information:

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

  - Operands next-use (later quadruple number)

  - Operands liveness (boolean)

- Symbolic `next-use` information

- Used in quadruple `g: x := y op z`:

- Mark `x` as dead in symbol table (previous value has no next use)

- Record `next` uses of `x`, `y` from symbol table into quadruple

- For quadruple `g: x := y op z`:

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

- Operands next-use (later quadruple number)

- Operands liveness (boolean)

- Each operand in a quadruple and symbol table carries additional information:

- Use symbol table to annotate status of variables

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

- Symbolic `next-use` information

- Removed of temporaries, for better register usage

- Interchange statements, for better scheduling

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

- Re-computation elimination: Recognize redundant computations

- Common subexpression elimination: Recognize redundant re-computation

- Symbolic `next-use` information

- Used in quadruple `g: x := y op z`:

- Mark `x` as dead in symbol table (previous value has no next use)

- Record `next` uses of `x`, `y` from symbol table into quadruple

- For quadruple `g: x := y op z`:

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

- Operands next-use (later quadruple number)

- Operands liveness (boolean)

- Each operand in a quadruple and symbol table carries additional information:

- Use symbol table to annotate status of variables

### Transformations on Basic Blocks

- Symbolic `next-use` information

- Used in quadruple `g: x := y op z`:

- Mark `x` as dead in symbol table (previous value has no next use)

- Record `next` uses of `x`, `y` from symbol table into quadruple

- For quadruple `g: x := y op z`:

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

- Operands next-use (later quadruple number)

- Operands liveness (boolean)

- Each operand in a quadruple and symbol table carries additional information:

- Use symbol table to annotate status of variables

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

- Symbolic `next-use` information

- Removed of temporaries, for better register usage

- Interchange statements, for better scheduling

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

- Re-computation elimination: Recognize redundant computations

- Common subexpression elimination: Recognize redundant re-computation

- Symbolic `next-use` information

- Used in quadruple `g: x := y op z`:

- Mark `x` as dead in symbol table (previous value has no next use)

- Record `next` uses of `x`, `y` from symbol table into quadruple

- For quadruple `g: x := y op z`:

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

- Operands next-use (later quadruple number)

- Operands liveness (boolean)

- Each operand in a quadruple and symbol table carries additional information:

- Use symbol table to annotate status of variables

### Transformations on Basic Blocks

- Symbolic `next-use` information

- Used in quadruple `g: x := y op z`:

- Mark `x` as dead in symbol table (previous value has no next use)

- Record `next` uses of `x`, `y` from symbol table into quadruple

- For quadruple `g: x := y op z`:

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

- Operands next-use (later quadruple number)

- Operands liveness (boolean)

- Each operand in a quadruple and symbol table carries additional information:

- Use symbol table to annotate status of variables
Register Allocation over Basic Block:

Register Allocation: 

Goal is to minimize use of registers and memory references.

Doubly linked data structure:

- For each variable, indicate location of current value: memory and/or registers:
  - address descriptor
- For each register, indicate current contents (set of variables with equal values):
  - register descriptor

Use getreg to obtain target register:

```
For x := yopz;
Call getreg to obtain target register R_i
Find current location of y, generate load into register R_i
Ditto for z, except that should use register other than R_i
```

On block exit, store registers that contain live values.

- Single load, register descriptor indicates that both x and y are live.
- For x := y,
  - Update address descriptor for x to indicate it resides in R_i
  - Update register descriptor for R_i to indicate it holds x.

- Emit instruction
  - 32-bit for x, except that should use register other than R_i
- If memory update address descriptor for y
- Find current location of y, generate load into register R_i in memory
- Call getreg to obtain target register R_i
- For x := y op z,
  - Emit instruction

Using getreg

```
Choose variable whose next use is farthest away.
Choose variable whose next use is farthest away.
```

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

```
For each variable, indicate location of current value:
- Variables with equal values: register descriptor
- For each register, indicate current contents (set of variables with equal values)
```

For quadruple x := y op z,

- For each register R_i
  - 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
  - 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
  - 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
  - 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

On block exit, store registers that contain live values.

```
For x := yopz;
Call getreg to obtain target register R_i
Find current location of y, generate load into register R_i
Ditto for z, except that should use register other than R_i
```

Using getreg

```
Choose variable whose next use is farthest away.
Choose variable whose next use is farthest away.
```

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

```
For each variable, indicate location of current value:
- Variables with equal values: register descriptor
- For each register, indicate current contents (set of variables with equal values)
```

For quadruple x := y op z,

- For each register R_i
  - 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
  - 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
  - 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
  - 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

On block exit, store registers that contain live values.

```
For x := yopz;
Call getreg to obtain target register R_i
Find current location of y, generate load into register R_i
Ditto for z, except that should use register other than R_i
```

Using getreg

```
Choose variable whose next use is farthest away.
Choose variable whose next use is farthest away.
```

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

```
For each variable, indicate location of current value:
- Variables with equal values: register descriptor
- For each register, indicate current contents (set of variables with equal values)
```

For quadruple x := y op z,
Computing Dependencies in a Basic Block:

**DAG Construction**

- Forward pass over basic block.
- Add x to list of labels of node which currently holds y.

**Example: dot product**

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

**Quadruples:**

- Basic block leader
- prod := 0
- Start: T1 := 4
- Loop:
  - prod := prod + a(T1) * b(T1); -- assume 4-byte integer
  - if j < 20 goto start
- End loop:
- prod := 0

**Common Subexpressions Identified**

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

**Intermediate Code Optimization:**

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

**Honors Compilers, NYU, Spring, 2007**

---

**DAG Construction**

- For x := y op z:
  - Start: T1 := 4
  - Loop:
    - prod := prod + a(T1) * b(T1); -- assume 4-byte integer
    - if j < 20 goto start
  - End loop:
    - prod := 0

**Example: dot product**

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

**Quadruples:**

- Basic block leader
- prod := 0
- Start: T1 := 4
- Loop:
  - prod := prod + a(T1) * b(T1); -- assume 4-byte integer
  - if j < 20 goto start
- End loop:
  - prod := 0

**Common Subexpressions Identified**

- Dag for body of loop.
- Add x to list of labels of node which currently holds y.
- For x := y.
  - Add y := z to list of labels of new node.
  - Create new node for "op", or find an existing one with descendants y, z (need hash scheme).
  - Find node labeled z, or create one.
  - Find node labeled y, or create one.
- For x := y op z.
  - Forward pass over basic block.
Register assignment is equivalent to graph coloring

Corresponding variables overlap:

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

Each variable is a node in the graph.

Lifetimes overlap.

Two variables cannot be assigned the same register if their

indefinite lifetimes need to know lifetimes of variables (set of

registers throughout

Optimal use of registers in subprograms: keep all variables in

Coloring

Using Global Information: Register

Coloring

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

The following requires 19 quadruples:

The following requires 19 quadruples:

end loop:

for k in lo2..hi2:


A exists in location:

components size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

From DAG to Improved Block

requires complex data structures and algorithms

Loop invariant computations

Live-dead analysis

Common subexpression elimination

Constant folding

Can compute global properties of program as iterative

Basic blocks are nodes in the flow graph

Fewer quadruples, fewer temporaries

If j := 20 goto start

A := T

B := T

prod := prod + T5;

j := j + 1;

if j = 20 goto start

A,B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!

A, B: array(0..1, hi2..lo2)

– component size w bytes

subexpression code generators don’t produce common

Programmers don’t produce common

subexpressions, code generators do!
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

Better Approach to Spilling

Use loop structure to estimate usage.

Spill variables with lowest usage count.

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

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

Example

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

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

\( R \) (no constraint) \( F \) \( D \) (constraint) \( E \) \( A \) \( F \) \( D \) \( E \) \( A \) \( F \) \( D \) \( E \) \( A \) \( F \) \( D \) \( E \) \( A \) \( F \) \( D \) \( E \) \( A \)

Fast heuristic algorithm (Chaitin) is usually linear:

Problem is NP-Complete

Assume 3 colors are available: assign colors in reverse order.

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

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

Spill variables with lowest usage count.

Use loop structure to estimate usage.

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.

Any node with fewer than \( N - 1 \) neighbors is colorable, so can be removed

Fast heuristic algorithm (Chaitin) is usually linear:

Problem is NP-Complete

Given a graph and a set of \( N \) colors, assign a color colored by the same color.