Optimizations

Hwansoo Han
Optimization

Idea

- Transform code into better (optimized) shape
  - For time, size, power, reliability, security, maintenance
- Eliminate redundancy in runtime execution for speed (time)

Scope to apply

- Local optimization is applied within a BB
- Global optimization is applied in the scope larger than BB
Simple Optimizations

- Several simple optimizations
  - Constant folding
  - Algebraic simplifications
  - Copy propagation
  - Constant propagation

- Commonly used as local optimizations
  - But can be extended to global optimizations, too.

Require no data-flow analysis

Require DFA for global optimization
Constant Folding

- Compile time computation for known values
  - Operations on constants can be computed at compile time
    - To decide if operands are constant across BB, DFA is needed
  - In general, if there is a statement \( x := y \ op \ z \)
    - And \( y \) and \( z \) are constants
    - Then \( y \ op \ z \) can be computed at compile time

- Examples
  - \( x = 2 + 2 \quad \rightarrow \quad x = 4 \)
  - \( \text{if} \ (2 < 0) \ \text{goto L} \) can be deleted
Algebraic Simplification

- Some statements can be deleted (identity operation)
  \[ x = x + 0, \quad x = x \times 1, \quad x = x \ll 0 \]
  \[ x = x | 0, \quad x = x & 0xffffffff \]

- Some statements can be simplified
  \[ x = x \times 0 \quad x = 0 \]
  \[ y = y ** 2 \quad y = y \times y \]
  \[ x = x \times 8 \quad x = x \ll 3 \]
  \[ x = x \times 15 \quad t = x \ll 4; \quad x = t - x \]
  (on some machines \( \ll \) is faster than \( \times \))
  \[ (x - y) + (x - y) \quad 2 \times x - 2 \times y \]
  (if \( x = 2^{31}, y = 2^{31} - 1 \), overflow occurs)
Copy Propagation

- Eliminate simple assignments
  - If \( w = x \) appears in a block,
    all subsequent uses of \( w \) can be replaced with uses of \( x \)

- Global copy propagation
  - Single assignment is important here.
  - In SSA, globally replace a copied variable with the other variable

Example:

\[
\begin{align*}
  b &= z + y \\
  a &= b \\
  x &= 2 \times a
\end{align*}
\]

\[
\begin{align*}
  b &= z + y \\
  a &= b \\
  x &= 2 \times b
\end{align*}
\]
Constant Propagation

- Eliminate simple assignment with a constant
  - Similar to copy propagation
  - If \( w = c \) appears in a block,
    all subsequent uses of \( w \) can be replaced with uses of \( c \)

- Global constant propagation
  - Using SSA, it is the same procedure as copy propagation

```
b = 3
c = 4 * b
If (c > b) goto L1

e = a + b
L1: d = b + 2
```

```
b = 3
c = 4 * 3
If (c > 3) goto L1

e = a + 3
L1: d = 3 + 2
```
Copy/Constant Propagation

- CP does not make the program smaller or faster, but might enable other optimizations
  - Constant folding
  - Dead code elimination
  - Algebraic simplification

Example:

```
a = 8
x = 2 * a
y = z * x
t = u
v = z + t
```

```
a = 8
x = 16
y = z << 4
v = z + u
```
Redundancy Elimination

- Unreachable code elimination
- Dead code elimination
- Common subexpression elimination
- Code motion
  - Loop invariant code motion
  - Partial redundancy elimination
  - Code hoisting
Unreachable Code Elimination

- Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional branch

- Why would such basic blocks occur?

- Removing unreachable code
  - Makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)
Dead Code Elimination

- Dead code created as a result of compiler optimizations
- If $w := rhs$ appears in a basic block and $w$ does not appear anywhere else in the program, then $w := rhs$ is dead and can be eliminated
  - Dead = does not contribute to the program’s result
    = not live right after definition

Example: (a is not used anywhere else)

\[
\begin{align*}
x &= z + y \\
a &= x \\
x &= 2 \ast a
\end{align*}
\]

SSA, CP

\[
\begin{align*}
x_1 &= z + y \\
a &= x_1 \\
x_2 &= 2 \ast x_1
\end{align*}
\]

DCE

\[
\begin{align*}
x_1 &= z + y \\
x_2 &= 2 \ast x_1
\end{align*}
\]
Common Subexpression Elimination

- CSE
  - Assume Basic block is in single assignment form
  - All assignments with same rhs compute the same value

- Example:

- Why is single assignment important here?
Dominance in CSE

Fully Redundant

OK, if b1 dom b2

Partially Redundant

NO, it needs PRE
CSE (cont’d)

- Global CSE
  - Solve “available expression” problem using DFA
    - Expression **must** be available from all paths at a joint point
  - Iterate statement within a basic block, maintaining “available expression” and replacing common expressions with a temporary variable (e.g. \( t_1 \) in the example)

\[
\begin{align*}
x &= a - b \\
&\ldots
\end{align*}
\]

\[
\begin{align*}
y &= a - b \\
&\ldots
\end{align*}
\]

\[
\begin{align*}
t_1 &= a - b \\
x &= t_1 \\
&\ldots
\end{align*}
\]

\[
\begin{align*}
y &= t_1 \\
&\ldots
\end{align*}
\]
Loop Invariant Code Motion

- Finding loop invariant
  - Operands are all constants
  - Operands are all defined outside the loop

- Calculating a loop invariant expression within a loop
  - Introduces redundant computation
  - Needs to be moved (hoisted) outside the loop

```
header
m = n + 2
n = 0

OK? NO
```

```
header
m = n + 2
n = 2
```
Loop Invariant Code Motion (cont’d)

- Hoist loop invariant code
  - Expressions are always safe, but assignments are not
    - Due to conditional execution or early exit, assignments might not execute each iteration even not execute at all
    - Could raise an exception, otherwise would not be raised
  - Assignment ( \( v = expression \) ) can be hoisted, if the following two conditions hold
    - The assignment dominates all uses of \( v \) in the loop, \textit{and}
    - The assignment dominates all the exit blocks of the loop
Partial Redundancy Elimination

- An expression is redundantly evaluated along some paths but not all paths
- Elimination (PRE)
  - Discover partial redundancies
  - Convert them to full redundancies
  - Remove them

\[
\begin{align*}
a &= x + y \\
b &= x + y \\
t &= x + y \\
a &= t \\
b &= t
\end{align*}
\]
Loop invariant code motion is actually a type of PRE

1\textsuperscript{st} iteration: \(x + y\) is not redundant, but

From the 2\textsuperscript{nd} iteration and on: \(x + y\) is redundant
Code Hoisting (Unification)

- If expressions are always evaluated after some point, move them to the *latest common dominator*

- Reduce code size, but execution time may not improve depending on dynamic factors – instr. scheduling, cache, etc.
Applying Optimizations

- Note that
  - Each optimization does very little by itself
  - Typically optimizations interact
  - Performing one optimizations enables other optimizations

- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - The optimizer can also be stopped at any time to limit the compilation time
Summary

- Eliminate redundancies!

- Local vs. global optimizations
  - Within BB vs. Across BB

- Optimizations
  - Simple optimizations that looks into a single statement
  - Optimizations that require DFA