Compiler Design

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
  - Value numbering
  - Copy propagation
  - Constant propagation

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

  \[
  \begin{align*}
  \text{Constant folding, Algebraic simplifications, Value numbering, Copy propagation} & \quad \text{Require no data-flow analysis} \\
  \text{Constant propagation} & \quad \text{Require DFA for global optimization}
  \end{align*}
  \]
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**
  - $x = 2 + 2 \quad x = 4$
  - if $(2 < 0)$ goto L can be deleted
  - $y = 2$
  - $y = 3$
  - $x = y + 2$
Algebraic Simplification

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

- **Some statements can be simplified**

  \[
  x = x \times 0 \quad \Rightarrow \quad x = 0
  \]
  
  \[
  y = y \times 2 \quad \Rightarrow \quad y = y \times y
  \]
  
  \[
  x = x \times 8 \quad \Rightarrow \quad x = x \ll 3
  \]
  
  \[
  x = x \times 15 \quad \Rightarrow \quad t = x \ll 4; \quad x = t - x
  \]

  (on some machines \(\ll\) is faster than \(\times\))

  \[
  (x - y) + (x - y) \quad \Rightarrow \quad 2 \times x - 2 \times y
  \]

  (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

L1: e = a + b

\[ d = b + 2 \]
```

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

L1: e = a + 3

\[ d = 3 + 2 \]
```
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  | a = 8 |
| x = 2 * a | x = 16 |
| y = z * x | y = z << 4 |
| t = u    | v = z + u |
| v = z + t |
```
Value Numbering

- **Eliminate computations if the equivalent values are already computed**
  - VN(n): assign an identifying number to each expression (n)
    - VN(x+y) = VN(j) iff x+y and j have the same value for all paths

- **Algorithm**
  - For each operation x = <operator, y, z>
    - Get VN for operands y and z from hash lookup
    - Hash <operator, VN(y), VN(z)> to get a VN for x
    - If x already had a VN, replace x with a reference of the VN
  - For each simple assignment x = y or x = const
    - Const/copy propagation is achieved by assigning the same VN
**Value Numbering (cont’d)**

- **Example in SSA**

<table>
<thead>
<tr>
<th>Original Code</th>
<th>With VNs</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0 = x_0 + y_0)</td>
<td>(a_0^3 = x_0^1 + y_0^2)</td>
<td>(a_0^3 = x_0^1 + y_0^2)</td>
</tr>
<tr>
<td>(b_0 = x_0 + y_0)</td>
<td>(\star b_0^3 = x_0^1 + y_0^2)</td>
<td>(b_0^3 = a_0^3)</td>
</tr>
<tr>
<td>(a_1 = 17)</td>
<td>(a_1^4 = 17)</td>
<td>(a_1^4 = 17)</td>
</tr>
<tr>
<td>(c_0 = x_0 + y_0)</td>
<td>(\star c_0^3 = x_0^1 + y_0^2)</td>
<td>(c_0^3 = a_0^3)</td>
</tr>
</tbody>
</table>

- **\(\star\):** redundant computation for the same value
- **RHS computation is replaced with the first reference computation**
- **Redundant expression is eliminated!**
Extensions to Value Numbering

- **Constant folding**
  - Add a bit in the hash to indicate a value is constant
  - Evaluate constant values at compile-time
  - Replace with load immediate or immediate operand

- **Algebraic identities**
  - List up special cases
    - Compute with identities
    - Computations that result in identities
  - Replace with input VN or immediate for the identity

- \( x+0, x-0, x*1, x/1, \)
- \( x-x, x*0, x\div x, x\sqrt{0}, x\land 0xFF...FF, \)
- \( \max(x,\text{MININT}), \)
- \( \min(x,\text{MAXINT}), \max(x,x), \)
- \( \min(y,y), \) and so on ...
Handling Larger Scopes - Superlocal VN

- **Extended Basic Blocks (EBB):** single entry, multiple exits
  - Initialize table for $b_i$ with table from $b_{i-1}$
  - With single-assignment naming, can use scoped hash table

The Plan:
- Process $b_1$, $b_2$, $b_4$
- **Pop** two levels
- Process $b_3$ relative to $b_1$
- **Start clean** with $b_5$
- **Start clean** with $b_6$
**Dominator Value Numbering**

- Can use hash table from \( \text{idom}(x) \) for basic block \( x \)
  - Use C for F, and A for G
  - Imposes a Dom-based application order
- Leads to **Dominator VN Technique (DVNT)**

### Dominator tree

- A
  - B
  - C
  - G
- D
  - E
  - F

```
A
  B
  C
  G
D
  E
  F
```
Global Value Numbering

- **To go further, we must deal with merge points**
  - Our simple naming scheme falls apart in $b_4$
  - We need more powerful analysis tools
  - Naming scheme becomes SSA

- **This requires global data-flow analysis**
  
  "Compile-time reasoning about the run-time flow of values"
  
  1. Build a model of control-flow
  2. Pose questions as sets of simultaneous equations
  3. Solve the equations
  4. Use solution to transform the code

Examples: LIVE, REACHES, AVAIL
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)

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

  SSA, CP

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

  DCE
Common Subexpression Elimination

- **CSE**
  - Assume Basic block is in single assignment form
  - All assignments with same \textit{rhs} compute the same value

- **Example:**

\[
\begin{align*}
x &= y + z \\
&\quad \ldots \\
w &= y + z
\end{align*}
\]

\[
\begin{align*}
x &= y + z \\
&\quad \ldots \\
w &= x
\end{align*}
\]

- **Why is single assignment important here?**
Dominance in CSE

\[ x = a - b \]
\[ z = a - b \]

\( a = 10 \)
\( b = 5 \)

\( z = x \)?

OK, if \( b_1 \) dom \( b_2 \)

Fully Redundant

\( b_0 \)
\( b_1 \)
\( b_2 \)

\( b_0 \)
\( b_1 \)
\( b_2 \)

\( b_0 \)
\( b_1 \)
\( b_2 \)

\( z = x \)?

NO, it needs PRE

Partially Redundant
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)

\[
\begin{align*}
  x &= a - b \\
  y &= a - b \\
  z &= a - b \\
  AVAIL &= \{a - b\} \\
\end{align*}
\]

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

\[
\begin{align*}
  t_1 &= a - b \\
  z &= t_1 \\
  AVAIL &= \{a - b\} \\
\end{align*}
\]
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

```
m = n + 2
n = 0
n = 2
OK? NO
```
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 ( \( v = \text{expression} \) ) can be hoisted, if the following two conditions hold
    - The assignment dominates all uses of \( v \) in the loop, 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
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**

- **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.

```
a < 10

e = c + d
d = 2
f = a + c
c + d > 0

g = a + c
h = a + c
exit
```

```
t1 = c + d
a < 10

e = t1
d = 2
f = t2
c + d > 0

g = t2
h = t2
exit
```
Loop Unrolling (1)

❖ **Most of exec-times are spent in loops**
  - Reduce loop overhead by unrolling the loop

```c
for (i=0; i<100; i++)
    a[i] = b[i] * c[i];
```

Complete unrolling

```c
a[0] = b[0] * c[0];
a[1] = b[1] * c[1];
...
a[99] = b[99] * c[99];
```

❖ Eliminated additions, tests, and branches
  - Only works with fixed loop bounds & few iterations
  - But increases the code size
❖ Unrolling is always safe, as long as we get the bounds right
Loop Unrolling (2)

- Unrolling by smaller factors is more practical

```c
for (i=0; i<100; i++)
    a[i] = b[i] * c[i];
```

Unrolling by 4:
```c
for (i=0; i<100; i+=4) {
    a[i]     = b[i]     * c[i];
    a[i+1]   = b[i+1]   * c[i+1];
    a[i+2]   = b[i+2]   * c[i+2];
    a[i+3]   = b[i+3]   * c[i+3];
}
```

- Achieves much of the benefits with lower code size growth
  - Reduces tests & branches by 25%
  - Less overhead per useful operation
  - Can apply local optimizations on the loop body

- But, it relied on knowledge of the loop bounds...
Loop Unrolling (3)

- **Unrolling with unknown bounds**
  - Need to generate guard loops

```
for (i=0; i<n; i++)
a[i] = b[i] * c[i];
```

```
for (i=0; i<n-3; i+=4)
  {
    a[i] = b[i] * c[i];
    a[i+1] = b[i+1] * c[i+1];
    a[i+2] = b[i+2] * c[i+2];
    a[i+3] = b[i+3] * c[i+3];
  }
for (; i<n; i++) // guard loop
  a[i] = b[i] * c[i];
```

- Achieves most of the benefits
  - Reduces tests & branches by 25%
- Guard loop takes extra code size
  - Can generalize to arbitrary upper & lower bounds, unroll factors
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

- **Optimization scope**
  - Local optimization: process within BB
  - Global optimization: process in the scope larger than BB

- **Value numbering as an example**
  - Local VN
  - Superlocal VN
  - Dominator VN
  - Global VN

- **Several other local optimization**
  - AS, CF, UCE, CSE, DCE, CP
  - All can be extended to global optimization with powerful analysis