Introduction to
Machine-Independent Optimizations - 6

Machine-Independent Optimization Algorithms

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2, 3, and 4)
- Fundamentals of control-flow analysis (in parts 4 and 5)
- Algorithms for machine-independent optimizations
- SSA form and optimizations
Mark as “invariant”, those statements whose operands are all either constant or have all their reaching definitions outside $L$

Repeat {
    Mark as “invariant” all those statements not previously so marked all of whose operands are constants, or have all their reaching definitions outside $L$, or have exactly one reaching definition, and that definition is a statement in $L$ marked “invariant”
} until no new statements are marked “invariant”
Loop Invariant Code motion Example

Before LIV code motion

```
t1 = 202
i = 1
L1: t2 = i > 100
   if t2 goto L2
   t1 = t1 - 2
   t3 = addr(a)
   t4 = t3 - 4
   t5 = 4*i
   t6 = t4 + t5
   *t6 = t1
   i = i + 1
   goto L1
L2:
```

After LIV code motion

```
t1 = 202
i = 1
   t3 = addr(a)
   t4 = t3 - 4
L1: t2 = i > 100
   if t2 goto L2
   t1 = t1 - 2
   t5 = 4*i
   t6 = t4 + t5
   *t6 = t1
   i = i + 1
   goto L1
L2:
```
Loop-Invariant Code Motion Algorithm

1. Find loop-invariant statements
2. For each statement $s$ defining $x$ found in step (1), check that
   (a) it is in a block that dominates all exits of $L$
   (b) $x$ is not defined elsewhere in $L$
   (c) all uses in $L$ of $x$ can only be reached by the definition of $x$ in $s$
3. Move each statement $s$ found in step (1) and satisfying conditions of step (2) to a newly created preheader
   provided any operands of $s$ that are defined in loop $L$ have previously had their definition statements moved to the preheader
The statement $i := 2$ from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4).
The computation gets altered due to code movement:

$i$ always gets value 2, and never 1, and hence $j$ always gets value 2

Condition 2(a):
s dominates all exits of $L$
The statement $i := 2$ from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4). The computation gets altered due to code movement: $i$ always gets value 2, and never 1, and hence $j$ always gets value 2.

Condition 2(a): $s$ dominates all exits of $L$.
Violation of condition 2(a) - Running Example

B1: \( i = 0 \)

B2:

- \( t_1 = \text{i} > 1 \)
- if \( \neg t_1 \) goto B9
- false

B3:

- \( j = 0 \)
- \( t_2 = i - 1 \)

B4:

- \( t_3 = j < t_2 \)
- if \( \neg t_3 \) goto B8
- false

B5:

- \( t_4 = 4 \times j \)
- \( t_5 = a[t_4] \)
- \( t_6 = j + 1 \)
- \( t_7 = 4 \times t_6 \)
- \( t_8 = a[t_7] \)
- \( t_9 = t_5 > t_8 \)
- if \( \neg t_9 \) goto B7
- false

B6:

- \( t_{11} = a[t_4] \)
- \( \text{temp} = t_{11} \)
- \( t_{13} = a + t_4 \)
- \( t_{16} = a[t_7] \)
- \( *t_{13} = t_{16} \)
- \( t_{19} = a + t_7 \)
- \( *t_{19} = \text{temp} \)

B7:

- \( j = t_6 \)
- goto B4

B8:

- \( i = t_2 \)
- goto B2

- stop

B9:

- B3 does not dominate B9
Code Motion - Violation of condition 2(b)

B2 dominates B4 and hence condition 2(a) is satisfied for i:=3 in B2. However, statement i:=3 from B2 cannot be moved to a preheader since condition 2(b) is violated (i is defined in B3).

The computation gets altered due to code movement:

*If the loop is executed twice, i may pass its value of 3 from B2 to j in the original loop.*

*In the revised loop, i gets the value 2 in the second iteration and retains it forever.*

**Condition 2(a):**

s dominates all exits of L

**Condition 2(b):**

x is not defined elsewhere in L
Code Motion - Violation of condition 2(c)

Conditions 2(a) and 2(b) are satisfied. However, statement
i:=2 from B4 cannot be moved to a preheader since condition
2(c) is violated (use of i in B6 is reached by defs of i in B1 and B4)

The computation gets altered due to code movement

*In the revised loop, i gets the value 2 from the def in the preheader and k becomes 2. However, k could have received the value of either 1 (from B1) or 2 (from B4) in the original loop*

Condition 2(a): s dominates all exits of L
Condition 2(b): x is not defined elsewhere in L
Condition 2(c): All uses of x in L can only be reached by the definition of x in s
Violation of condition 2(c) - Running Example

B2 dominates B9, but \( j+1 \) in B5 is reached by \( j=0 \) and \( j=t_6 \)

- B1: \( i = 0 \)
- B2: \( j = 0 \), \( t_1 = i > 1 \)
- B3: \( t_2 = i - 1 \)
- B4: \( t_3 = j < t_2 \)
- B5: \( t_4 = 4^*j \), \( t_5 = a[t_4] \), \( t_6 = j + 1 \), \( t_7 = 4^*t_6 \), \( t_8 = a[t_7] \), \( t_9 = t_5 > t_8 \)
- B6: \( t_{11} = a[t_4] \), \( \text{temp} = t_{11} \), \( t_{13} = a + t_4 \), \( t_{16} = a[t_7] \), \( *t_{13} = t_{16} \), \( t_{19} = a + t_7 \), \( *t_{19} = \text{temp} \)
- B7: \( j = t_6 \), goto B4

Y.N. Srikant  Optimization Algorithms
The Static Single Assignment Form:
Application to Program Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- SSA form - definition and examples
- Optimizations with SSA forms
  - Dead-code elimination
  - Simple constant propagation
  - Copy propagation
  - Conditional constant propagation and constant folding
The SSA Form: Introduction

- A new intermediate representation
- Incorporates \textit{def-use} information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a \textit{static} single assignment form, and not a \textit{dynamic} single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A \textit{sparse} intermediate representation
  - If a variable has \(N\) uses and \(M\) definitions, then \textit{def-use chains} need space and time proportional to \(N \cdot M\)
  - But, the corresponding instructions of uses and definitions are only \(N + M\) in number
  - SSA form, for most realistic programs, is linear in the size of the original program
A Program in non-SSA Form and its SSA Form

read $A, B, C$
if $(A>B)$
    if $(A>C)$ $max = A$
else $max = C$
else if $(B>C)$ $max = B$
else $max = C$
printf $(max)$

Y.N. Srikant
Program Optimizations and the SSA Form
A program is in SSA form, if each use of a variable is reached by exactly one definition.
Flow of control remains the same as in the non-SSA form.
A special merge operator, \( \phi \), is used for selection of values in join nodes.
Not every join node needs a \( \phi \) operator for every variable.
No need for a \( \phi \) operator, if the same definition of the variable reaches the join node along all incoming edges.
Often, an SSA form is augmented with \( u-d \) and \( d-u \) chains to facilitate design of faster algorithms.
Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency.
Program 2 in non-SSA and SSA Form

- **Start**
- i = 0
  - i = i + 1
  - even(n)
    - n = n/2
    - n = 3*n + 1
    - Stop

- n <= 1
  - B2

- **B1**
  - read n

- **B3**
  - even(n)
    - n = n/2
    - n = 3*n + 1

- **B4**
  - print i

- **B5**
  - i = i + 1

---

- **Start**
- i_1 = 0
  - read n_1

- **B1**
  - i_2 = \Phi(i_3, i_1)
    - n_2 = \Phi(n_5, n_1)
    - n_2 <= 1

- **B2**
  - even(n_2)
    - print i_2

- **B3**
  - even(n_2)
    - n_3 = n_2/2

- **B4**
  - print i_2

- **B5**
  - i_3 = i_2 + 1

- **B6**
  - n_4 = 3*n_2 + 1

- **B7**
  - n_5 = \Phi(n_3, n_4)
  - Stop

Y.N. Srikant
Program Optimizations and the SSA Form
Program 3 in non-SSA and SSA Form

**Non-SSA Form:**
- **B0**: Start
- **B1**: Read A, LSR = 1; RSR = A, SR = (LSR + RSR) / 2
- **B2**: T = SR × SR
- **B3**: T > A
- **B4**: RSR = SR
- **B5**: LSR = SR
- **B6**: RSR = SR
- **B7**: LSR = SR
- **B8**: SR = (LSR + RSR) / 2
- **B9**: LSR! = RSR
- **B10**: Print SR
- **B11**: Stop

**SSA Form:**
- **B0**: Start
- **B1**: Read A, LSR₁ = 1; RSR₁ = A, SR₁ = (LSR₁ + RSR₁) / 2
- **B2**: LSR₂ = \( \Phi(LSR₅, LSR₁) \)
- **B3**: RSR₂ = \( \Phi(RSR₅, RSR₁) \)
- **B4**: SR₂ = \( \Phi(SR₃, SR₁) \)
- **B5**: T = SR₂ × SR₂
- **B6**: LSR₃ = SR₂
- **B7**: LSR₄ = SR₂
- **B8**: RSR₅ = \( \Phi(LSR₂, LSR₃, LSR₄) \)
- **B9**: RSR₅ = \( \Phi(RSR₃, RSR₂, RSR₄) \)
- **B10**: Print SR₃
- **B11**: Stop
Dead-code elimination

- Very simple, since there is exactly one definition reaching each use
- Examine the du-chain of each variable to see if its use list is empty
- Remove such variables and their definition statements
- If a statement such as $x = y + z$ (or $x = \phi(y_1, y_2)$) is deleted, care must be taken to remove the deleted statement from the du-chains of $y$ and $z$ (or $y_1$ and $y_2$)

Simple constant propagation

Copy propagation

Conditional constant propagation and constant folding

Global value numbering
Simple Constant Propagation

\[
\{ \text{Stmtpile} = \{ S | S \text{ is a statement in the program} \} \\
\text{while Stmtpile is not empty} \{ \\
\quad S = \text{remove}(\text{Stmtpile}); \\
\quad \text{if } S \text{ is of the form } x = \phi(c, c, \ldots, c) \text{ for some constant } c \\
\quad \quad \text{replace } S \text{ by } x = c \\
\quad \text{if } S \text{ is of the form } x = c \text{ for some constant } c \\
\quad \quad \text{delete } S \text{ from the program} \\
\quad \quad \text{for all statements } T \text{ in the } du\text{-chain} \text{ of } x \text{ do} \\
\quad \quad \quad \text{substitute } c \text{ for } x \text{ in } T; \text{ simplify } T \\
\quad \text{Stmtpile} = \text{Stmtpile} \cup \{ T \} \\
\}\}
\]

Copy propagation is similar to constant propagation

- A single-argument \( \phi \)-function, \( x = \phi(y) \), or a copy statement, \( x = y \) can be deleted and \( y \) substituted for every use of \( x \)
SSA forms along with extra edges corresponding to \( d-u \) information are used here
- Edge from every definition to each of its uses in the SSA form (called henceforth as SSA edges)

Uses both flow graph and SSA edges and maintains two different work-lists, one for each (\textit{Flowpile} and \textit{SSApile}, resp.)

Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values

Flow graph edges are added to \textit{Flowpile}, whenever a branch node is symbolically executed or whenever an assignment node has a single successor
SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node.

This ensures that all uses of a definition are processed whenever a definition changes its lattice value.

This algorithm needs much lesser storage compared to its non-SSA counterpart.

Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to true or false) or both edges (corresponding to ⊥) are added to the worklist.

However, at any join node, the meet operation considers only those predecessors which are marked executable.
CCP Algorithm - Example 1 - Trace 1

Y.N. Srikant
Program Optimizations and the SSA Form
CCP Algorithm - Example 1 - Trace 2

Y. N. Srikant

Program Optimizations and the SSA Form
CCP Algorithm - Example 2

Start

B1
- a1 = 1
- b1 = 1
- c1 = 0

B2
- b2 = \Phi(b4, b1)
- c2 = \Phi(c4, c1)
- if c2 < 100

B3
- if b2 < 20

B4
- Stop

B5
- true
- b3 = a1
- c3 = c2 + 1

B6
- false
- b5 = c2
- c5 = c2 + 1

B7
- b4 = \Phi(b3, b5)
- c4 = \Phi(c3, c5)