Introduction to Machine-Independent Optimizations - 7
Program Optimizations and the SSA Form

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 (in part 6)
- SSA form and optimizations
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

Conditional constant propagation is faster and more effective on SSA forms
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 ($Flowpile$ and $SSApile$, 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 $Flowpile$, 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 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
true
false

B4
Stop

B5
b3 = a1
c3 = c2 + 1

B6
b5 = c2
c5 = c2 + 1

B7
b4 = \Phi(b3, b5)
c4 = \Phi(c3, c5)
CCP Algorithm - Example 2 - Trace 1

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

false

B6

b5 = c2
c5 = c2 + 1

B7

b4 = \Phi(b3, b5)
c4 = \Phi(c3, c5)
CCP Algorithm - Example 2 - Trace 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

false

B6

b5 = c2
c5 = c2 + 1

B7

b4 = \Phi(b3, b5)
c4 = \Phi(c3, c5)
CCP Algorithm - Example 2 - Trace 3

B1

\[
\begin{align*}
a_1 &= 1 \\
b_1 &= 1 \\
c_1 &= 0
\end{align*}
\]

Start

B2

\[
\begin{align*}
b_2 &= \Phi(b_1) = 1 \\
c_2 &= \Phi(c_1) = 0 \\
\text{if } c_2 < 100: & \text{ true}
\end{align*}
\]

true

B3

if \( b_2 < 20 \)

false

B4

Stop

B5

\[
\begin{align*}
b_3 &= a_1 \\
c_3 &= c_2 + 1
\end{align*}
\]

false

B6

\[
\begin{align*}
b_5 &= c_2 \\
c_5 &= c_2 + 1
\end{align*}
\]

true

B7

\[
\begin{align*}
b_4 &= \Phi(b_3, b_5) \\
c_4 &= \Phi(c_3, c_5)
\end{align*}
\]
CCP Algorithm - Example 2 - Trace 4

Start

B1: a1 = 1, b1 = 1, c1 = 0

B2: b2 = \Phi(b1) = 1, c2 = \Phi(c1) = 0
if c2 < 100: true

B3: if b2 < 20: true

B4: Stop

B5: b3 = a1, c3 = c2 + 1

B6: b5 = c2, c5 = c2 + 1

B7: b4 = \Phi(b3, b5), c4 = \Phi(c3, c5)
CCP Algorithm - Example 2 - Trace 5

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = \Phi(b1) = 1
c2 = \Phi(c1) = 0
if c2 < 100: true

B3

if b2 < 20: true

B5

b3 = a1 = 1
c3 = c2 + 1 = 1

B6

b5 = c2
c5 = c2 + 1

B7

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

Stop

Y.N. Srikant
Program Optimizations and the SSA Form
CCP Algorithm - Example 2 - Trace 6

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = Φ(b1) = 1
c2 = Φ(c1) = 0
if c2 < 100: true

B3

if b2 < 20: true

B5

b3 = a1 = 1
c3 = c2 + 1 = 1

B6

b5 = c2
c5 = c2 + 1

B7

b4 = Φ(b3) = 1
c4 = Φ(c3) = 1

Stop

second visit
CCP Algorithm - Example 2 - Trace 7

Start

B1

\[
\begin{align*}
\text{a1} &= 1 \\
\text{b1} &= 1 \\
\text{c1} &= 0
\end{align*}
\]

B2

second visit, change in value of c2; no change in value of b2

\[
\begin{align*}
\text{b2} &= \Phi(\text{b4}, \text{b1}) = 1 \\
\text{c2} &= \Phi(\text{c4}, \text{c1}) = 1 \\
\text{if c2 < 100: unknown}
\end{align*}
\]

B3

true

\[
\begin{align*}
\text{if b2 < 20: true}
\end{align*}
\]

B4

false

Stop

B5

true

\[
\begin{align*}
\text{b3} &= \text{a1} = 1 \\
\text{c3} &= \text{c2} + 1 = 1
\end{align*}
\]

B6

false

\[
\begin{align*}
\text{b5} &= \text{c2} \\
\text{c5} &= \text{c2} + 1
\end{align*}
\]

B7

\[
\begin{align*}
\text{b4} &= \Phi(\text{b3}) = 1 \\
\text{c4} &= \Phi(\text{c3}) = 1
\end{align*}
\]
CCP Algorithm - Example 2 - Trace 8

Start

B1

a1 = 1  
b1 = 1  
c1 = 0

B2

b2 = \Phi(b4, b1) = 1  
c2 = \Phi(c4, c1) = \bot  
if c2 < 100: unknown

B3

if b2 < 20: true

B4

Stop

B5

true

B6

false

B7

b3 = a1 = 1  
c3 = c2 + 1 = \bot

b4 = \Phi(b3) = 1  
c4 = \Phi(c3) = 1

b5 = c2  
c5 = c2 + 1
CCP Algorithm - Example 2 - Trace 9

Start

B1

\[ a_1 = 1 \]
\[ b_1 = 1 \]
\[ c_1 = 0 \]

B2

\[ b_2 = \Phi(b_4, b_1) = 1 \]
\[ c_2 = \Phi(c_4, c_1) = \bot \]
if \( c_2 < 100 \): unknown

true

B3

if \( b_2 < 20 \): true

false

B4

Stop

B5

\[ b_3 = a_1 = 1 \]
\[ c_3 = c_2 + 1 = \bot \]

B6

Nothing happens in B6 because it is not reachable by a flow edge

B7

\[ b_4 = \Phi(b_3) = 1 \]
\[ c_4 = \Phi(c_3) = 1 \]
CCP Algorithm - Example 2 - Trace 10

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = \Phi(b4, b1) = 1
c2 = \Phi(c4, c1) = \bot
if c2 < 100: unknown

B3

if b2 < 20: true

B5

true

b3 = a1 = 1
c3 = c2 + 1 = \bot

false

B6

b5 = c2
c5 = c2 + 1

B7

b4 = \Phi(b3) = 1
c4 = \Phi(c3) = \bot

Stop
CCP Algorithm - Example 2 - Trace 11

Start

B1

\( a_1 = 1 \)
\( b_1 = 1 \)
\( c_1 = 0 \)

B2

\( b_2 = \Phi(b_4, b_1) = 1 \)
\( c_2 = \Phi(c_4, c_1) = \perp \)

if \( b_2 < 100 \): unknown

third visit to B2, no change in either b2 or c2; algorithm stops

B3

if \( b_2 < 20 \): true

true

B5

\( b_3 = a_1 = 1 \)
\( c_3 = c_2 + 1 = \perp \)

false

B4

Stop

false

B6

\( b_5 = c_2 \)
\( c_5 = c_2 + 1 \)

B7

\( b_4 = \Phi(b_3) = 1 \)
\( c_4 = \Phi(c_3) = \perp \)
After first round of simplification

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = 1
c2 = \Phi(c4, c1)
if c2 < 100

B5

b3 = 1
c3 = c2 + 1

B7

b4 = 1
c4 = \Phi(c3) = c3

Start

false

Stop

B4
Start

B2

\[ c_2 = \Phi(c_3, 0) \]
if \( c_2 < 100 \)

B5

true

c3 = c2+1

false

Stop

B4

After second round of simplification – elimination of dead code, elimination of trivial \( \Phi \)-functions, copy propagation etc.
Instruction Scheduling and Software Pipelining - 1

Y.N. Srikant

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

NPTEL Course on Principles of Compiler Design
Instruction Scheduling
  - Simple Basic Block Scheduling
  - Trace, Superblock and Hyperblock scheduling

Software pipelining
Instruction Scheduling

- Reordering of instructions so as to keep the pipelines of functional units full with no stalls
- NP-Complete and needs heuristics
- Applied on basic blocks (local)
- Global scheduling requires elongation of basic blocks (super-blocks)
Instruction Scheduling - Motivating Example

- time: load - 2 cycles, op - 1 cycle
- This code has 2 stalls, at i3 and at i5, due to the loads

(a) Sample Code Sequence

| i1  | r1 ← load a |
| i2  | r2 ← load b |
| i3  | r3 ← r1 + r2 |
| i4  | r4 ← load c |
| i5  | r5 ← r3 - r4 |
| i6  | r6 ← r3 * r5 |
| i7  | d ← st r6 |

(b) DAG
Scheduled Code - no stalls

- There are no stalls, but dependences are indeed satisfied

| i1: | r1 ← load a |
| i2: | r2 ← load b |
| i4: | r4 ← load c |
| i3: | r3 ← r1 + r2 |
| i5: | r5 ← r3 - r4 |
| i6: | r6 ← r3 * r5 |
| i7: | d ← st r6 |
Consider the following code:

\[ i_1 : r1 \leftarrow \text{load}(r2) \]
\[ i_2 : r3 \leftarrow r1 + 4 \]
\[ i_3 : r1 \leftarrow r4 + r5 \]

The dependences are

- \( i_1 \delta i_2 \) (flow dependence)
- \( i_2 \overline{\delta} i_3 \) (anti-dependence)
- \( i_1 \delta^0 i_3 \) (output dependence)

anti- and output dependences can be eliminated by register renaming
Dependence DAG

- full line: *flow* dependence, dash line: *anti*-dependence
- dash-dot line: *output* dependence
- some anti- and output dependences are because memory disambiguation could not be done

---

| i1   | t1 ← load a |
| i2   | t2 ← load b |
| i3   | t3 ← t1 + 4 |
| i4   | t4 ← t1 - 2 |
| i5   | t5 ← t2 + 3 |
| i6   | t6 ← t4 * t2 |
| i7   | t7 ← t3 + t6 |
| i8   | c ← st t7   |
| i9   | b ← st t5   |

(a) Instruction Sequence

(b) DAG
Basic Block Scheduling

- Basic block consists of micro-operation sequences (MOS), which are indivisible.
- Each MOS has several steps, each requiring resources.
- Each step of an MOS requires one cycle for execution.
- Precedence constraints and resource constraints must be satisfied by the scheduled program.
  - PC’s relate to data dependences and execution delays.
  - RC’s relate to limited availability of shared resources.
The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, $G = (V, E)$
  - $R$: number of resources
  - Nodes ($V$): MOS; Edges ($E$): Precedence
  - Label on node $v$
    - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with $v$
    - length $l(v)$ of node $v$
  - Label on edge $e$: Execution delay of the MOS, $d(e)$

Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that
\[ \forall e = (u, v) \in E, \sigma(v) - \sigma(u) \geq d(e) \] and
\[ \forall i, \sum_{v \in V} \rho_v(i - \sigma(v)) \leq R, \] where
the length of the schedule is
\[ \max_{v \in V} \{\sigma(v) + l(v)\} \]
Consider R = 5. Each MOS substep takes 1 time unit.

- At \( i=4 \), \( c_{v4}(1) + c_{v3}(2) + c_{v2}(3) + c_{v1}(4) = 2+2+1+0 = 5 \leq R \), satisfied
- At \( i=2 \), \( c_{v3}(0) + c_{v2}(1) + c_{v1}(2) = 3+3+2 = 8 > R \), NOT satisfied
A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

FUNCTION ListSchedule (V,E)
BEGIN
    Ready = root nodes of V; Schedule = $\phi$;
    WHILE Ready $\neq \phi$ DO
        BEGIN
            v = highest priority node in Ready;
            $Lb = \text{SatisfyPrecedenceConstraints} (v, \text{Schedule}, \sigma)$;
            $\sigma(v) = \text{SatisfyResourceConstraints} (v, \text{Schedule}, \sigma, Lb)$;
            Schedule = Schedule + {$v$};
            Ready = Ready - {$v$} + {$u | \text{NOT} (u \in \text{Schedule})$}
            AND $\forall (w, u) \in E, w \in \text{Schedule}$;
        END
    RETURN $\sigma$;
END
FUNCTION SatisfyPrecedenceConstraint(v, Sched, σ)
BEGIN
    RETURN ( \[\max_{u \in Sched} \sigma(u) + d(u, v)\])
END

FUNCTION SatisfyResourceConstraint(v, Sched, σ, Lb)
BEGIN
    FOR i := Lb TO \[\infty\] DO
        IF \[\forall 0 \leq j < l(v), \rho_v(j) + \sum_{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R\] THEN
            RETURN (i);
        END
    END
Precedence Constraint Satisfaction

Lower bound for $\sigma(v) = 29$

Already scheduled nodes

Node to be scheduled

Precedence constraint satisfaction:

$v$ can be scheduled only after all of $u_1$, $u_2$, and $u_3$, finish

Lower bound for $\sigma(v)$

$= \max(10+2, 25+4, 18+3)$

$= \max(12, 29, 21) = 29$
Resource constraint satisfaction

Consider $R = 5$. Each MOS substep takes 1 time unit.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Time $\sigma(u)$</th>
<th>MOS substeps (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma(v_1)=0$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\sigma(v_2)=1$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$\sigma(v_3)=4$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$\sigma(v_4)=5$</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Time slots 2 and 3 are vacant because scheduling node $v_3$ in either of them violates resource constraints.