Static Single Assignment Form

CS426

October 31, 1996

Jaejin Lee
Department of Computer Science
jlee@csrd.uiuc.edu
What is Static Single Assignment (SSA) form?

For the last few years, static single assignment (SSA) form has been proposed to represent data flow and control flow properties of sequential programs and being used in optimizing compilers. The SSA form has shown its usefulness for powerful code optimizations that are very efficient in terms of time and space.

Two properties of SSA form

- There is only one reaching definition for each use of a variable.

- The code contains $\phi$-functions that distinguish values of variables coming from different control flow edges.
Why SSA form?

• Easy to reason about variables.
  – if two variables have the same name they also contain the same value.

• Compact def-use chains.
  – If there are $d$ definitions and $u$ uses for a variable, there can be $d \times u$ def-use chains. In SSA form, there are at most $E$ def-use chains, where $E =$ the number of edges in the CFG.
  – Easy to generate def-use chains. Don’t need to solve data flow problem with bit-vectors in order to get reaching definitions.
Where to use SSA form?

- Constant propagation
- Common subexpression elimination
- Partial redundancy elimination
- Code motion
- Induction variable analysis
- Instruction scheduling (avoids output- and antidependences when there is no loop)
References


An example of SSA form

a = 4
b = 5
if P then
    a = a + b
else
    a = a - b
endif

Other

c = a + 1

\[ a_0 = 4 \]
\[ b_0 = 5 \]
if P then
    a_1 = a_0 + b_0
else
    a_2 = a_0 - b_0
endif

a_3 = φ(a_1, a_2)

\[ c_1 = a_3 + 1 \]
An example of constant propagation

\[
\begin{bmatrix}
\top & \top & C_i & C_j & \bot \\
\top & \bot & C_i & C_i & \bot \\
C_i & \bot & C_i & \bot & \bot \\
\bot & \bot & \bot & \bot & \bot \\
\end{bmatrix}
\]

\[
i = 1 \\
j = 2 \\
k = 0 \\
if \ i = 1 \ then \\
\quad j = i + 2 \\
else \\
\quad j = i + 3 \\
endif \\
k = i + j \\
print \ k
\]

Entry

\[
i = 1 \\
j = 2 \\
k = 0
\]

\[
i = 1
\]

\[
j = i + 2 \quad j = i + 3
\]

\[
k = i + j \\
print \ k
\]

Exit
An example of constant propagation (continue)

```plaintext
i0=1  
j0=2  
k0=0  

i0=1

j1=i0+2  
j3=i0+3

j2=\phi (j1, j3)  
kl=i0+j2  
print kl

j2=\phi (j1, j3)  
kl=i0+j2  
print kl

k1=4  
print 4
```

```plaintext
i0=1  
j0=2  
k0=0  
j1=3  
k1=4  
print 4
```
An example of partial redundancy elimination

if P then
    y = ...
else
    x = ...
    = x + y
endif
= x + y

if P then
    y = ...
else
    x = ...
    = x + y
endif
= x + y 
endif
Definitions

Definition 1 A $\phi$-assignment has the form $U=\phi(V,W,\ldots)$ where $U,V,W,\ldots$ are variables and the number of operands $V,W,\ldots$ is the number of control flow predecessors of the node where the $\phi$-assignment occurs. $\phi(V,W,\ldots)$ is called a $\phi$-function.

Definition 2 For any nonnegative integer $n$, a path of length $n$ in CFG consists of a sequence $x_0,\ldots,x_n$ of $n+1$ nodes and a sequence $e_1,\ldots,e_n$ of $n$ edges such that $e_i$ goes from $x_{i-1}$ to $x_i$ for all $i$ with $1 \leq i \leq n$. We write $e_i$ as $x_{i-1} \rightarrow x_i$. The null path is the one when $n = 0$. We write $p:x_0 \rightarrow^* x_n$ when there is a path from $x_0$ to $x_n$, but $p:x_0 \rightarrow^+ x_n$ if $p$ is anonnull path.
Definitions (continue)

**Definition 3** Nonnull paths \( p : x_0 \rightarrow^+ x_n \) and \( q : y_0 \rightarrow^+ y_m \) converge at a node \( z \) if

\[
(x_0 \neq y_0) \\
\wedge (x_n = y_m = z) \\
\wedge ((x_i = y_j) \Rightarrow (i = n \lor j = m))
\]

**Definition 4** Given a set \( S \) of CFG nodes, the set \( J(S) \) of join nodes is the set of all nodes \( z \) such that there are two nonnull CFG paths that start at two distinct nodes in \( S \) and converge at \( z \).

**Definition 5** Given a set \( S \) of CFG nodes, the iterated join \( J^+(S) \) is the limit of the increasing sequence of sets of nodes

\[
J_1 = J(S) \\
J_{i+1} = J(S \cup J_i)
\]
Definitions (continue)

**Definition 6** If a node $x$ appears on every path from Entry to a node $y$, then $x$ **dominates** $y$. If $x$ dominates $y$ and $x \neq y$, then $x$ **strictly dominates** $y$.

**Definition 7** The dominance frontier $DF(x)$ of a CFG node $x$ is the set of all CFG nodes $y$ such that $x$ dominates a predecessor of $y$ but does not strictly dominate $y$. Given a set $S$ of CFG nodes, the dominance frontier $DF(S)$ of the set $S$ is defined by $DF(S) = \bigcup_{x \in S} DF(x)$.

**Definition 8** The iterated dominance frontier $DF^+(S)$ is the limit of the increasing sequence of sets of nodes

$$DF_1 = DF(S)$$
$$DF_{i+1} = DF(S \cup DF_i)$$
Where to place $\phi$-functions?

**Theorem 1** *For any set $S$ of CFG nodes,*

$$ J^+(S) \subseteq DF^+(S) $$

**Theorem 2** *For any set $S$ of CFG nodes such that Entry $\in S,*$

$$ DF^+(S) \subseteq J^+(S) $$

**Theorem 3** *The set of nodes that need $\phi$-functions for any variable $v$ is the iterated dominance frontier $DF^+(S)$, where $S$ is the set of nodes with assignments to $v.*

**Basic idea**

1. Place $\phi$-functions

2. Rename variable
Arrays

- The entire array is treated like a single scalar variable

- **Access(A,i)**: *i*th component of *A*

- **Update(A,j,V)**: an array value that is of the same size as *A* and has the same component values as *A* other than *j*th component which has the value *V*

\[
\begin{align*}
\ldots &= A(i) \\
A(j) &= V \\
\ldots &= A(k) + 2 \\
\end{align*}
\]

\[
\begin{align*}
\ldots &= \text{Access}(A,i) \\
A &= \text{Update}(A,j,V) \\
T &= \text{Access}(A,k) \\
\ldots &= T + 2 \\
\end{align*}
\]

\[
\begin{align*}
\ldots &= \text{Access}(A_1,i_1) \\
A_2 &= \text{Update}(A_1,j_5,V_4) \\
T_1 &= \text{Access}(A_2,k_2) \\
\ldots &= T_1 + 2 \\
\end{align*}
\]
Why do we calculate iterated join?

In order to find out the place where $\phi$-assignments are placed.

Why do we calculate iterated dominance frontier?

The calculation of iterated join is inefficient.

However, for structured programs, we don’t even need to calculate iterated dominance frontier. Join nodes are known at parse time.
Single-Pass Generation of SSA Form

• Generation of SSA form at parse time.

• Can be done on Control Flow Graph as well.

• Restricted to structured programs. IF, CASE, WHILE, REPEAT, FOR, DO, etc. but no GOTO.

• Join nodes are known a priory
Control structures and their join nodes

IF, CASE

WHILE, FOR, DO

REPEAT

PROCEDURE

Entry

Exit

Dummy join node for nested REPEAT

REPEAT

body

UNTIL P

UNTIL Q
Naming of values (variables)

- Every assignment to a variable $v$ generates a new value $v_i$ where $i$ is a unique number.

- Just after the assignment to $v$, $v_i$ is the current value of $v$.

- Every subsequent use of $v$ is replaced by its current value.
<table>
<thead>
<tr>
<th>Source code</th>
<th>SSA form</th>
<th>Current values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = 0$</td>
<td>$v_1 = 0$</td>
<td>$v_0 \ x_0$</td>
</tr>
<tr>
<td>$x = v + 1$</td>
<td>$x_1 = v_1 + 1$</td>
<td>$v_1 \ x_0$</td>
</tr>
<tr>
<td>$v = 2$</td>
<td>$v_2 = 2$</td>
<td>$v_1 \ x_1$</td>
</tr>
</tbody>
</table>

**Modification of a $\phi$-assignment when compiling**
Compiling IF statement

IF cond THEN
  a = 1
  b = a + 1
ELSE
  a = a + 1
  c = 2
ENDIF

d = a

...
Compiling WHILE statement

\[ a = 1 \]
\[ \text{WHILE } \text{cond} \text{ DO} \]
\[ b = a + 1 \]
\[ a = a \times 2 \]
\[ \text{ENDWHILE} \]
\[ d = a \]
\[ \ldots \]
Compiling REPEAT statement

IF cond1 THEN
a = 1
REPEAT
b = b + a
UNTIL cond2
ENDIF
d = b
...

1
2
3
4

1

2

3

4

a = 1
b = b + a
cond1
d = b

a0
a1 = 1
b = b + a
cond1
b1

a2 = φ (a1)
d = b

1 2
3 4

a0, b0

a1 = 1
b0
b2 = φ (b0, b1)
b1 = b2 + a1
cond1
b1

a2 = φ (a1)  
a0
b3 = φ (b1)  
b0
d = b

a0
a0
a1
a2 = φ (a1, a0)  
a0
b3 = φ (b1, b0)  
b0
d1 = b3
Exercise

I = 1
J = 1
K = 1
L = 1
REPEAT
  IF (P) THEN
    J = I
    IF (Q) THEN
      L = 2
    ELSE
      L = 3
    ENDIF
    K = K + 1
  ELSE
    K = K + 2
  ENDIF
  PRINT(I,J,K,L)
UNTIL (T)