Static single assignment

**SSA with dominance property**
- Unique definition for each variable.
- Each definition dominates its uses.
Static single assignment

SSA with dominance property

- Unique definition for each variable.
- Each definition dominates its uses.

\[ a = \ldots \]
\[ b = \ldots \]
\[ n = b \]
\[ b = a \]
\[ a = n \]
Static single assignment

SSA with dominance property
- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA
- Need to introduce $\phi$-functions at the (iterated) dominance frontier.

\[
\begin{align*}
B_0 & : a = \ldots \\
B_1 & : n = b \\
& \quad b = a \\
& \quad a = n
\end{align*}
\]
**Static single assignment**

**SSA with dominance property**
- Unique definition for each variable.
- Each definition dominates its uses.

**Conversion into SSA**
- Need to introduce $\phi$-functions at the (iterated) dominance frontier.

\[
\begin{align*}
B_0 & : \\
& a_1 = \ldots \\
& b_1 = \ldots \\
B_1 & : \\
& a_2 = \phi(a_1, a_3) \\
& b_2 = \phi(b_1, b_3) \\
& n = b_2 \\
& b_3 = a_2 \\
& a_3 = n
\end{align*}
\]
Static single assignment

SSA with dominance property

- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce $\phi$-functions at the (iterated) dominance frontier.

Interests of SSA

- Link uses/definitions explicit.
- Code optimizations: efficient, easy-to-implement, fast.
- More accurate program analysis.
Static single assignment

SSA with dominance property
- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA
- Need to introduce $\phi$-functions at the (iterated) dominance frontier.

Interests of SSA
- Link uses/definitions explicit.
- Code optimizations: efficient, easy-to-implement, fast.
- More accurate program analysis.
Dominance can be computed by fixed-point iteration:

\[ D(r) = \{r\} \text{ and } D(n) = \{n\} \cup \left( \bigcap_{p \in \text{pred}[n]} D[p] \right) \]

Many other more efficient algorithms are possible. Then:

```plaintext
procedure computeDF(n)
    S := {}
    for each node y in succ[n] do
        if (idom(y) \neq n) then S := S \cup \{y\} /* successor of n not strictly dominated by n */
    endfor
    for each child c of n in the dominator tree do
        computeDF(c)
        for each element w of DF[c] do
            if (n does not dominate w) then S := S \cup \{w\}
        endfor
    endfor
    DF[n] := S
```

Alain Darte  Cours M2: Compilation avancée et optimisation de programmes
What is the dominance frontier of node 5?
Example

First we must find all nodes that node 5 strictly dominates.
Example

A node $w$ is in the dominance frontier of node 5 if 5 dominates a predecessor of $w$, but 5 does not strictly dominates $w$ itself. What is the dominance frontier of 5?
Example

DF(5) = \{4, 5, 12, 13\}

A node \( w \) is in the dominance frontier of node 5 if 5 dominates a predecessor of \( w \), but 5 does not strictly dominates \( w \) itself. What is the dominance frontier of 5?
procedure Place-ϕ-functions(G, DF, D) /* D[n] is the set of variables defined in n */
for each node n in G do
    for each variable a in D[n] do
        defsites[a] := defsites[a] ∪ {n}
    endfor
endfor
for each variable a do
    W := defsites[a]
    while (W not empty) do
        remove some node n from W
        for each Y in DF[n] do
            if (Y ∉ D_{ϕ}[n]) then
                insert statement a = ϕ(a, . . . , a) at the top of Y
                D_{ϕ}[n] := D_{ϕ}[n] ∪ {Y}
                if (Y ∉ D[n]) then W := W ∪ {Y}
            endif
        endfor
    endwhile
endfor
procedure Rename(n)
   for each statement S in block n do
      if (S is not a φ-function) then
         for each use of some variable x in S do
            i := top(Stack[x]); replace the use of x with x_i in S
         endfor
      endif
   for each definition of some variable a in S
      Count[a] += 1; i := Count[a]; push i onto Stack[a]; replace definition with a_i
   endfor
   for each successor Y of block n and each φ-function in Y do
      i := top(Stack[a]) where a is the argument coming from n; replace it with a_i
   endfor
   for each child (in the dominance tree) X of n do Rename(X)
   endfor
   for each definition of some variable a (in the original code) do pop Stack[a]
end procedure

procedure RenameAll(G)
   for each variable a do Count[a] := 0; Stack[a] := {}; push 0 onto Stack[a]
   Rename(r) /* root of the dominance tree */
SSA: A Complete Example.

```plaintext
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
k=k+1;
    }
    else {
        j=k;
k=k+2;
    }
} return j;
```
SSA: A Complete Example.

\[
i = 1; \\
j = 1; \\
k = 0; \\
\textbf{while}(k < 100) \{ \\
\quad \textbf{if}(j < 20) \{ \\
\quad \quad j = i; \\
\quad \quad k = k + 1; \\
\quad \} \\
\quad \textbf{else} \{ \\
\quad \quad j = k; \\
\quad \quad k = k + 2; \\
\quad \} \\
\} \\
\textbf{return} j;
\]
SSA: A Complete Example.

```c
i=1;
j=1;
k=0;
while(k<100) {
  if(j<20) {
    j=i;
k=k+1;
  } else {
    j=k;
k=k+2;
  }
}
return j;
```
i=1; 
j=1; 
k=0; 
while(k<100) {
    if(j<20) {
        j=i; 
        k=k+1;
    } 
    else {
        j=k; 
        k=k+2;
    }
} 
return j;

j ← φ(j_0,j) 
if j<20 
    j ← φ(j_0,j) 
    k ← φ(k_0,k) 
    if k<100 
    j ← i_0 
    k ← k+1 
    return j 

j ← k 
    k ← k+2 

j ← φ(j,j) 
    k ← φ(k,k)
SSA: A Complete Example.

\[ i = 1; \]
\[ j = 1; \]
\[ k = 0; \]
\[ \textbf{while} (k < 100) \{ \]
\[ \quad \textbf{if} (j < 20) \{ \]
\[ \quad \quad j \leftarrow i; \]
\[ \quad \quad k \leftarrow k + 1; \]
\[ \quad \}\]
\[ \quad \textbf{else} \{ \]
\[ \quad \quad j \leftarrow k; \]
\[ \quad \quad k \leftarrow k + 2; \]
\[ \quad \}\]
\[ \}\]
\[ \textbf{return} \ j; \]
SSA: A Complete Example.

```plaintext
i=1;
j=1;
k=0;
while(k<100) {
  if(j<20) {
    j=i;
k=k+1;
  } else {
    j=k;
k=k+2;
  }
}
return j;
```

```
B1: i_0 ← 1
j_0 ← 1
k_0 ← 0

B2: j_1 ← φ(j_0,j_3)
k_1 ← φ(k_0,k_3)
if k_1<100

B3: if j_1<20

B4: return j_1

B5: j_2 ← i_0
k_2 ← k_1+1

B6: j ← k
k ← k+2

B7: j_3 ← φ(j_2,j)
k_3 ← φ(k_2,k)
```
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
k=k+1;
    } else {
        j=k;
k=k+2;
    }
}
return j;
```
Example: Constant Propagation

B1: $i_0 \leftarrow 1$
    $j_0 \leftarrow 1$
    $k_0 \leftarrow 0$

B2: $j_1 \leftarrow \phi(j_0, j_3)$
    $k_1 \leftarrow \phi(k_0, k_3)$
    if $k_1 < 100$

B3: if $j_1 < 20$
    return $j_1$

B4: $j_2 \leftarrow i_0$
    $k_2 \leftarrow k_1 + 1$
    $j_4 \leftarrow k_1$
    $k_4 \leftarrow k_1 + 2$

B5: $j_3 \leftarrow \phi(j_2, j_4)$
    $k_3 \leftarrow \phi(k_2, k_4)$

B6: $j_1 \leftarrow \phi(1, j_3)$
    $k_1 \leftarrow \phi(0, k_3)$
    if $k_1 < 100$

B7: $j_2 \leftarrow 1$
    $k_2 \leftarrow k_1 + 1$
    $j_4 \leftarrow k_1$
    $k_4 \leftarrow k_1 + 2$

B8: $j_3 \leftarrow \phi(j_2, j_4)$
    $k_3 \leftarrow \phi(k_2, k_4)$
Example:
Dead-code Elimination

B1

\[ i_0 \leftarrow 1 \]
\[ j_0 \leftarrow 1 \]
\[ k_0 \leftarrow 0 \]

\[ j_1 \leftarrow \phi(1,j_3) \]
\[ k_1 \leftarrow \phi(0,k_3) \]
if \( k_1 < 100 \)

B2

B3

if \( j_1 < 20 \)

return \( j_1 \)

j_2 \leftarrow 1
k_2 \leftarrow k_1 + 1

j_4 \leftarrow k_1
k_4 \leftarrow k_1 + 2

B4

B5

B6

j_3 \leftarrow \phi(j_2,j_4)
k_3 \leftarrow \phi(k_2,k_4)

B7

if \( j_1 < 20 \)

return \( j_1 \)

j_2 \leftarrow 1
k_2 \leftarrow k_1 + 1

j_4 \leftarrow k_1
k_4 \leftarrow k_1 + 2

CMPUT 680 - Compiler Design and Optimization
Constant Propagation and Dead Code Elimination

\[ j_1 \leftarrow \phi(1,j_3) \]
\[ k_1 \leftarrow \phi(0,k_3) \]
if \( k_1 < 100 \)

\[ j_2 \leftarrow 1 \]
\[ k_2 \leftarrow k_1 + 1 \]

\[ j_4 \leftarrow k_1 \]
\[ k_4 \leftarrow k_1 + 2 \]

\[ j_3 \leftarrow \phi(j_2,j_4) \]
\[ k_3 \leftarrow \phi(k_2,k_4) \]

if \( j_1 < 20 \)
return \( j_1 \)

if \( k_1 < 100 \)

return \( j_1 \)
Example: Is this the end?

But block 6 is never executed! How can we find this out, and simplify the program?

SSA conditional constant propagation finds the least fixed point for the program and allows further elimination of dead code.

Example: Dead code elimination

\[
\begin{align*}
    j_1 & \leftarrow \phi(1,j_3) \\
    k_1 & \leftarrow \phi(0,k_3) \\
    \text{if } k_1 & < 100 \\
    j_1 & \leftarrow \phi(1,j_3) \\
    k_1 & \leftarrow \phi(0,k_3) \\
    \text{if } k_1 & < 100 \\
    j_3 & \leftarrow \phi(1,j_4) \\
    k_3 & \leftarrow \phi(k_2,k_4) \\
    j_4 & \leftarrow k_1 \\
    k_4 & \leftarrow k_1 + 2
    \end{align*}
\]
Example: Single Argument \( \phi \)-Function Elimination

\[
\begin{align*}
B2: & \quad j_1 \leftarrow \phi(1,j_3) \\
& \quad k_1 \leftarrow \phi(0,k_3) \\
& \quad \text{if } k_1 < 100
\end{align*}
\]

B4: \quad return \( j_1 \)

B7: \quad j_3 \leftarrow 1 \\
& \quad k_3 \leftarrow k_2
\]

B2: \quad j_1 \leftarrow \phi(1,j_3) \\
& \quad k_1 \leftarrow \phi(0,k_3) \\
& \quad \text{if } k_1 < 100
\]

B4: \quad return \( j_1 \)

B7: \quad j_3 \leftarrow \phi(1) \\
& \quad k_3 \leftarrow \phi(k_2)
\]
Example: Constant and Copy Propagation

\[ j_1 \leftarrow \phi(1, j_3) \]
\[ k_1 \leftarrow \phi(0, k_3) \]
\[ \text{if } k_1 < 100 \]
\[ j_3 \leftarrow 1 \]
\[ k_3 \leftarrow k_2 \]

\[ k_2 \leftarrow k_1 + 1 \]

return \( j_1 \)

\[ j_1 \leftarrow \phi(1, 1) \]
\[ k_1 \leftarrow \phi(0, k_2) \]
\[ \text{if } k_1 < 100 \]
\[ j_3 \leftarrow 1 \]
\[ k_3 \leftarrow k_2 \]

return \( j_1 \)
Example:
Dead Code Elimination

\[
\begin{align*}
    j_1 & \leftarrow \phi(1,1) \\
    k_1 & \leftarrow \phi(0,k_2) \\
    \text{if } k_1 & < 100
\end{align*}
\]

return \( j_1 \)

\[
\begin{align*}
    k_2 & \leftarrow k_1 + 1
\end{align*}
\]

\[
\begin{align*}
    j_3 & \leftarrow 1 \\
    k_3 & \leftarrow k_2
\end{align*}
\]

CMPUT 680 - Compiler Design and Optimization
Example:

**ϕ-Function Simplification**

\[
\begin{align*}
j_1 & \leftarrow \phi(1,1) \\
k_1 & \leftarrow \phi(0,k_2) \quad \text{if } k_1 < 100 \\
\text{return } j_1 \\
k_2 & \leftarrow k_1 + 1
\end{align*}
\]
Example: Constant Propagation

\[ j_1 \leftarrow 1 \]
\[ k_1 \leftarrow \phi(0, k_2) \text{ if } k_1 < 100 \]
\[ \text{return } j_1 \]
\[ k_2 \leftarrow k_1 + 1 \]

\[ j_1 \leftarrow 1 \]
\[ k_1 \leftarrow \phi(0, k_2) \text{ if } k_1 < 100 \]
\[ \text{return } 1 \]
\[ k_2 \leftarrow k_1 + 1 \]
Example:
Dead Code Elimination

```
return 1

B4

k_2 \leftarrow k_1 + 1

B4

return 1
```

```
j_1 \leftarrow 1
k_1 \leftarrow \phi(0,k_2)
if k_1 < 100
```

```
return 1
```

```
k_1 \leftarrow \phi(0,k_2)
if k_1 < 100
```

```
return 1
```
References


Recent advances in SSA

- SSA-based compilers & JIT compilation.
- Register allocation, out-of-SSA conversion, liveness analysis.
- SSA extensions: SSI, gated SSA, psi-SSA, value state dependence graph, array SSA, safeTSA, etc.
A few important results:

- If $S$ contains the entry node, $J(S) = J^+(S) = DF^+(S)$.
- $G$ is reducible
  - iff simplifiable by the rules $T_1$ and $T_2$.
  - iff each SCC has a unique entry node.
  - iff removing all $(u, v)$ where $v$ dominates $u$ makes $G$ acyclic.
  - ...  
- Dominators and iterated dominance frontiers can be computed quickly from loop-nesting forest, especially if $G$ is reducible.
- Conversely, DJ-graphs can be used to build loop forests.
- Advanced algorithms use Tarjan’s union-find with almost-linear complexity (see Ramalingam, Sreedhar, Havlak, Steensgaard).
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.

**Swap problem**

\[
\begin{align*}
a_1 &= \ldots \\
b_1 &= \ldots \\
B_0 &
\end{align*}
\]

\[
\begin{align*}
a_2 &= \phi(a_1, a_3) \\
b_2 &= \phi(b_1, b_3) \\
n &= b_2 \\
b_3 &= a_2 \\
a_3 &= n \\
B_1 &
\end{align*}
\]
Early attempts and pitfalls

- **Cytron et al. (1991):** copies in predecessor basic blocks.

\[
\begin{align*}
B_0 &: \\
& a_1 = \ldots \\
& b_1 = \ldots \\
B_1 &: \\
& a_2 = \phi(a_1, b_2) \\
& b_2 = \phi(b_1, a_2)
\end{align*}
\]
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.

\[
\begin{align*}
B_0: & \quad a_1 = \ldots \\
& \quad b_1 = \ldots \\
& \quad a_2 = a_1 \\
& \quad b_2 = b_1 \\
B_1: & \quad a_2 = \phi(a_1, b_2) \\
& \quad b_2 = \phi(b_1, a_2) \\
& \quad a_2 = b_2 \\
& \quad b_2 = a_2
\end{align*}
\]

Many SSA optimizations turned off in gcc and Jikes.

Alain Darte

Cours M2: Compilation avancée et optimisation de programmes
Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
- Bad understanding of parallel copies.

\[
\begin{align*}
B_0: & \quad a_1 = \ldots \\
& \quad b_1 = \ldots \\
& \quad a_2 = a_1 \\
& \quad b_2 = b_1 \\
B_1: & \quad a_2 = b_2 \\
& \quad b_2 = a_2
\end{align*}
\]
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

Lost copy problem

\[
\begin{align*}
B_0 & \quad x = \ldots \\
B_1 & \quad y = x \\
& \quad x = x + 1 \\
\end{align*}
\]
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

Lost copy problem

\[
\begin{align*}
B_0: & \quad x_1 = \ldots \\
B_1: & \quad x_2 = \phi(x_1, x_3) \\
& \quad y = x_2 \\
& \quad x_3 = x_2 + 1 \\
\end{align*}
\]
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

Lost copy problem

\[
\begin{align*}
B_0: & \\
& x_1 = \ldots \\
B_1: & \\
& x_2 = \phi(x_1, x_3) \\
& x_3 = x_2 + 1 \\
& x_2 \\
\end{align*}
\]
Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
- Bad understanding of parallel copies.

Lost copy problem

\[
\begin{align*}
B_0 & : \quad x_1 = \ldots \\
B_1 & : \quad x_2 = \phi(x_1, x_3) \\
& \quad x_3 = x_2 + 1 \\
& \quad x_2 = x_3 \\
& \quad x_2
\end{align*}
\]
Early attempts and pitfalls

- **Cytron et al. (1991):** copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

Lost copy problem

\[ x_1 = \ldots \]
\[ x_2 = x_1 \]

\[ x_3 = x_2 + 1 \]
\[ x_2 = x_3 \]
\[ x_2 \]

Many SSA optimizations turned off in gcc and Jikes.
Early attempts and pitfalls

- **Cytron et al. (1991):** copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

- **Briggs et al. (1998):** both problems identified. General correctness unclear.

\[
\begin{align*}
  x_1 &= \ldots \\
  B_0 &\quad x_2 = x_1 \\
  B_1 &\quad x_3 = x_2 + 1 \\
  &\quad x_2 = x_3 \\
  &\quad x_2
\end{align*}
\]
Early attempts and pitfalls

- **Cytron et al. (1991):** copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

- **Briggs et al. (1998):** both problems identified. General correctness unclear.

- **Sreedhar et al. (1999):** correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - “virtualization” hard to implement.

---

**Lost copy problem**

```
\begin{align*}
B_0: & \quad x_1 = \ldots \\
       & \quad x_2 = x_1 \\
B_1: & \quad x_3 = x_2 + 1 \\
       & \quad x_2 = x_3 \\
x_2: & \\
\end{align*}
```
Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - “virtualization” hard to implement.

Many SSA optimizations turned off in gcc and Jikes.
Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all $\phi$-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

From SSA to CSSA

```
B_0

B_1

B_i

B_n
```

Liveness of $\phi$ defined by the $a_i'$.

Be careful with potential bugs due to conditional branches that use or define variables.
Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all $\phi$-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

Correctness

After introduction of variables $a'_i$ and copies, the code is in CSSA.

From SSA to CSSA

$B_1$

\[
\begin{align*}
a'_1 &= a_1
\end{align*}
\]

$B_i$

\[
\begin{align*}
a'_i &= a_i
\end{align*}
\]

$B_n$

\[
\begin{align*}
a'_n &= a_n
\end{align*}
\]

$B_0$

\[
\begin{align*}
a'_0 &= \phi(a'_1, \ldots, a'_n) \\
a_0 &= a'_0
\end{align*}
\]
Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all $\phi$-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

Correctness

After introduction of variables $a'_i$ and copies, the code is in CSSA.

Code quality

Aggressive coalescing can remove useless copies. But better use accurate notion of interferences.

From SSA to CSSA

\[
\begin{align*}
B_1 & : a'_1 = a_1 \\
B_i & : a'_i = a_i \\
B_n & : a'_n = a_n \\
B_0 & : a'_0 = \phi(a'_1, \ldots, a'_n) \\
& \quad a_0 = a'_0
\end{align*}
\]

“Liveness of $\phi$” defined by the $a'_i$.

† Be careful with potential bugs due to conditional branches that use or define variables.
Coalesced example: the swap problem

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]

\[ a_2 = \phi(a_1, b_2) \]
\[ b_2 = \phi(b_1, a_2) \]

\[ (u_0, v_0) = (a_1, b_1) \]
\[ u_0 = \phi(u_1, u_2) \]
\[ v_0 = \phi(v_1, v_2) \]
\[ (a_2, b_2) = (u_0, v_0) \]
\[ (u_2, v_2) = (b_2, a_2) \]
Coalesced example: the swap problem

\[
\begin{align*}
  a_1 &= \ldots \\
  b_1 &= \ldots \\
  B_0 &\quad (u_1, v_1) = (a_1, b_1) \\
  B_1 &\quad (a_2, b_2) = (u_0, v_0) \\
  (u_2, v_2) &= (b_2, a_2)
\end{align*}
\]
Coalesced example: the swap problem

\[
a_1 = \ldots \\
b_1 = \ldots \\
(u_1, v_1) = (a_1, b_1)
\]

\[
B_0
\]

\[
a_1 = \ldots \\
b_1 = \ldots \\
(u_1, v_1) = (a_1, b_1)
\]

\[
B_1
\]

\[
u_0 = \phi(u_1, u_2) \\
v_0 = \phi(v_1, v_2) \\
(a_2, b_2) = (u_0, v_0) \\
(u_2, v_2) = (b_2, a_2)
\]
Coalesced example: the swap problem

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]

\[ B_0 \]

\[ (u_2, v_2) = (b_2, a_2) \]

\[ B_1 \]

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]

\[ (u_1, v_1) = (a_1, b_1) \]

\[ B_0 \]

\[ u_0 = \phi(u_1, u_2) \]
\[ v_0 = \phi(v_1, v_2) \]
\[ (a_2, b_2) = (u_0, v_0) \]

\[ (u_2, v_2) = (b_2, a_2) \]

\[ B_1 \]
Coalesced example: the swap problem

\[
\begin{align*}
B_0 &: a_1 = \ldots, \quad b_1 = \ldots \\
B_1 &:
\begin{cases}
(u_2, v_2) = (b_2, a_2) \\
(n = b_1, b_1 = a_1, a_1 = n)
\end{cases}
\end{align*}
\]
Coalesced example: the lost copy problem

\[ x_1 = \ldots \]

\[ B_0 \]

\[ x_2 = \phi(x_1, x_3) \]
\[ x_3 = x_2 + 1 \]

\[ B_1 \]

\[ x_2 \]

\[ B_0 \]

\[ u_0 = \phi(u_1, u_2) \]
\[ x_2 = u_0 \]
\[ x_3 = x_2 + 1 \]
\[ u_2 = x_3 \]

\[ B_1 \]

\[ x_2 \]
Coalesced example: the lost copy problem

\[
x_1 = \ldots
u_1 = x_1
\]

\[
B_0
\]

\[
x_2 = u_0
x_3 = x_2 + 1
u_2 = x_3
\]

\[
B_1
\]

\[
u_0 = \phi(u_1, u_2)
\]
Coalesced example: the lost copy problem

\[ u = (u_0, u_1, u_2) \]

\[ x_1 = \ldots \]

\[ u_1 = x_1 \]

\[ B_0 \]

\[ x_2 = u_0 \]

\[ x_3 = x_2 + 1 \]

\[ u_2 = x_3 \]

\[ B_1 \]

\[ x_1 = \ldots \]

\[ u_1 = x_1 \]

\[ u_0 = \phi(u_1, u_2) \]

\[ x_2 = u_0 \]

\[ x_3 = x_2 + 1 \]

\[ u_2 = x_3 \]
Coalesced example: the lost copy problem

\begin{align*}
B_0 & \quad x_1 = \ldots \\
B_1 & \quad x_2 = x_1 \\
& \quad x_1 = x_2 + 1 \\
\end{align*}

\begin{align*}
B_0 & \quad u_1 = x_1 \\
B_1 & \quad u_0 = \phi(u_1, u_2) \\
& \quad x_2 = u_0 \\
& \quad x_3 = x_2 + 1 \\
& \quad u_2 = x_3 \\
\end{align*}
Outline

1. Code representations
   - Control-flow graph
   - Loop-nesting forest
   - Static single assignment

2. Out-of-SSA translation
   - Translation with copy insertions: pitfalls and solution
   - Improving code quality and ease of implementation
   - Fast implementation with reduced memory footprint

3. SSA properties and liveness
   - Dominance, liveness, interferences, and chordal graphs
   - Construction of liveness sets in reducible CFGs for strict SSA
   - Extensions to irreducible CFGs and for checking liveness
Exploiting SSA: value-based interferences

Definition (Chaitin interference)
Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

\[
b = \ldots \quad c = \ldots \quad a = \phi(b,c) \quad d = \phi(b,a)
\]
Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

\[ b = \ldots \]
\[ a' = b \]
\[ d' = b \]

\[ c = \ldots \]
\[ a' = c \]

\[ a = a' \]
\[ d' = a \]

\[ d = d' \]
Exploiting SSA: value-based interferences

**Definition (Chaitin interference)**

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.
Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

Need to update interference graph after coalescing.
Exploiting SSA: value-based interferences

**Definition (Chaitin interference)**

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

Need to update interference graph after coalescing.

**Unique value \( V \) of a SSA variable**

For a copy \( b = a \), \( V(b) = V(a) \) (traversal of dominance tree).

**Value-based interference**

\( a \) and \( b \) interfere if \( V(a) \neq V(b) \) and \( \text{Live-range}(a) \cap \text{Live-range}(b) \neq \emptyset \).
Exploiting SSA: value-based interferences

**Definition (Chaitin interference)**
Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

- Need to update interference graph after coalescing.

**Unique value $V$ of a SSA variable**
For a copy $b = a$, $V(b) = V(a)$ (traversal of dominance tree).

**Value-based interference**
a and $b$ interfere if $V(a) \neq V(b)$ and $\text{Live-range}(a) \cap \text{Live-range}(b) \neq \emptyset$. 

![Diagram showing value-based interference](image)
Using parallel copies instead of sequential copies

Parallel copy semantics

\[ (a_1, \ldots, a_n) = (b_1, \ldots, b_n), \text{ all copies } a_i = b_i \text{ are simultaneous.} \]

- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.
Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

(a, b, c, d) = (c, a, b, c)
Using parallel copies instead of sequential copies

Parallel copy semantics

\[
(a_1, \ldots, a_n) = (b_1, \ldots, b_n), \text{ all copies } a_i = b_i \text{ are simultaneous.}
\]
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges \( b_i \to a_i \).
- Directed trees with roots = circuits.
- Insert copies for the leaves first.
Using parallel copies instead of sequential copies

Parallel copy semantics

In \((a_1, \ldots, a_n) = (b_1, \ldots, b_n)\), all copies \(a_i = b_i\) are simultaneous.

- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges \(b_i \rightarrow a_i\).

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

\[
\begin{align*}
d &= c \\
c &= b \\
b &= a \\
a &= d
\end{align*}
\]
Using parallel copies instead of sequential copies

Parallel copy semantics

In \((a_1, \ldots, a_n) = (b_1, \ldots, b_n)\), all copies \(a_i = b_i\) are simultaneous.

- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges \(b_i \rightarrow a_i\).

- Directed trees with roots=circuits.
- Insert copies for the leaves first.
- Simple circuit: one more copy.
**Algorithm 1:** Parallel copy sequentialization algorithm

**Data:** Set $P$ of parallel copies $a \mapsto b$, $a \neq b$, one extra fresh variable $n$

**Output:** List of copies in sequential order

1. $\text{ready} \leftarrow \emptyset$; $\text{to}\_\text{do} \leftarrow \emptyset$; $\text{pred}(n) \leftarrow \bot$
2. $\textbf{forall the } (a \mapsto b) \in P \textbf{ do}$
   - $\text{loc}(b) \leftarrow \bot$; $\text{pred}(a) \leftarrow \bot$; /* initialization */
3. $\textbf{forall the } (a \mapsto b) \in P \textbf{ do}$
   - $\text{loc}(a) \leftarrow a$; $\text{pred}(b) \leftarrow a$; $\text{to}\_\text{do}.\text{push}(b)$; /* copy into $b$ to be done */
4. $\textbf{forall the } (a \mapsto b) \in P \textbf{ do}$
   - $\text{if } \text{loc}(b) = \bot \text{ then } \text{ready}.\text{push}(b)$; /* $b$ is not used and can be overwritten */
5. $\textbf{while } \text{to}\_\text{do} \neq \emptyset \textbf{ do}$
   - $\text{while ready } \neq \emptyset \textbf{ do}$
     - $b \leftarrow \text{ready}.\text{pop}()$; $a \leftarrow \text{pred}(b)$; /* pick a free location */
     - $c \leftarrow \text{loc}(a)$; $\text{emit}\_\text{copy}(c \mapsto b)$; $\text{loc}(a) \leftarrow b$; /* generate the copy */
     - $\text{if } a = c \text{ and } \text{pred}(a) \neq \bot \text{ then } \text{ready}.\text{push}(a)$; /* first time copied */
   - $b \leftarrow \text{to}\_\text{do}.\text{pop}()$; /* look for remaining copy */
   - $\textbf{if } b = \text{loc}(b) \textbf{ then}$
     - $\text{emit}\_\text{copy}(b \mapsto n)$; $\text{loc}(b) \leftarrow n$; $\text{ready}.\text{push}(b)$; /* break circuit */
Qualitative experiments with SPEC CINT2000

Key points of the out-of-SSA translation

- Copy insertion (to go to CSSA and to handle register renaming constraints) followed by coalescing.
- Value-based interferences coalescing is improved and independent of virtualization (i.e., as in Sreedhar III).
- Parallel copies followed by sequentialization.
Bug tracking RVM-254 of Jikes RVM

Problems with SSA form: lack of loop unrolling breaks VM

This problem is probably one of the most serious in the RVM currently. When loop unrolling is disabled and SSA enabled the created IR is corrupt. The error has in the past look like we were suffering from the "lost copy" problem, but implementing a naive solution to this didn’t solve the problem. Their is sound logic behind the code so we need to identify a small test case where things are broken and then reason about what's wrong in leave SSA. This has been attempted once (with the code that removes an element from the live set) but the problem no longer appears to surface here. Currently these optimizations are disabled but by RVM 3.0 they should be re-enable and this bug cured.
Potential bugs with conditional branches

Initial code:
- $B_0$
  - $u = \ldots$
  - $v = \ldots$
- $B_1$
- $B_2$
- $B_3$
- $B_4$
- $w = \phi(u, v)$
- $\ldots = w$

"Blind" Sreedhar III:
- $B_0$
  - $u = \ldots$
  - $v = \ldots$
- $B_1$
- $B_2$
- $B_3$
- $B_4$
- $v' = v$
- $\phi(u, B_3, B_4)$
- $w = \phi(u, v')$
- $\ldots = w$

Wrong output code:
- $B_0$
  - $w = \ldots$
  - $v = \ldots$
- $B_1$
- $B_2$
- $B_3$
- $B_4$
- $w = v$
- $\phi(w, B_3, B_4)$
- $\ldots = w$
Unfeasible out-of-SSA translation example

Initial code

After optimization

Needs edge splitting

\[
\begin{align*}
B_1 & : \quad u_1 = \phi(u_0, u_2) \\
& \quad u_2 = u_1 - 1 \\
& \quad t_0 = u_2 \\
& \quad Br(u_2, B_1, B_2) \\
B_2 & : \quad t_1 = \phi(t_0, t_2) \\
& \quad t_2 = t_1 + \ldots \\
& \quad Br(t_2, B_1, B_2) \\
B_3 & : \quad \ldots = u_2
\end{align*}
\]

\[
\begin{align*}
B_1 & : \quad Br_{\text{dec}}(u, B_1, B_2) \\
B_2 & : \quad t_1 = \phi(t_0, t_2) \\
& \quad t_2 = t_1 + \ldots \\
& \quad Br(t_2, B_1, B_2) \\
B_3 & : \quad \ldots = u
\end{align*}
\]

\[
\begin{align*}
B_1 & : \quad t_0 = u \\
B_2 & : \quad t_1 = \phi(t_0, t_2) \\
& \quad t_2 = t_1 + \ldots \\
& \quad Br(t_2, B_1, B_2) \\
B_3 & : \quad \ldots = u
\end{align*}
\]