263-2810: Advanced Compiler Design

4.3.1 Strict SSA

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland
Example program (sketch)

```plaintext
if (...) {
    if (...) {
        a = 0
    } else {
        a = 1
    }
}

b = a * f

} else {
...
}
```
Example program (sketch)

```java
if (...) {
    if (...) {
        a = 0
    } else {
        a = 1
    }
} else {
    b = a * f
} else {
    ...
}
```
If \( a \) is dead outside the then-block, then there is no need to reconcile versions

- After outer then-else
- \( \phi \) functions are (often) not free!

Unfortunately the algorithm in 3.4 did not consider liveness information
4.3.1 Strict SSA

- Strict SSA: $\phi$ functions for a variable $V$ are inserted only into blocks where $V$ is live
  - In the basic blocks determined by approach outlined in Section 3.4
  - 3.4 yields strict SSA if all variables are live till EXIT

- Compiler must know liveness information
Liveness information

- There exist (at least) 3 different approaches to collect liveness information
  1. Dataflow analysis
  2. On-the-fly collection
  3. Split data space: local vs global
Dataflow analysis

- **Solve dataflow equations**
  - E.g., as discussed in basic compiler class(es)
  - Bit vector framework

- **Extra effort**
  - One of the motivations for SSA is to get rid of bit vector-based flow analysis

- **Solutions to dataflow equations are an approximation**
On-the-fly collection

- Algorithm in Section 3.5 processes all basic blocks
- Collect liveness information as compiler identifies versions for variables
  - If a variable $V$ appears on the RHS, identify version $V_j$ that is used and link all occurrences to definition
  - Def-use chains
Split data space

- Approximate liveness information by keeping a bit for each variable (version) if it used outside the basic block that defines the variable
  - Insert ϕ function only if variable is used in another basic block
  - Benefits of (strict) SSA for all variables defined and used inside one basic block
- Won’t catch example on previous slide
Example program (sketch)

```plaintext
if (...) {
    if (...) {
        a = 0
    } else {
        a = 1
    }
} else {
    b = a * f
} else {
    ...
}
```

```
if (…) {
    if (…) {
        a = 0
    } else {
        a = 1
    }
} else {
    b = a * f
} else {
    ...
}
```
4.3.2 De-SSA

- The CFG with $\phi$ functions must (at some time) be turned into format handled by code generator
  - Any code generator
  - Code generator of student compiler

- $\phi$ functions capture global properties
  - Code generator deals with local properties
  - Don’t want to burden code generator with handling $\phi$ functions

- No processor’s instruction set contains $\phi$ instructions
Translation from SSA form \(\rightarrow\) conventional IR

- Basic idea: insert copies

\[a_1 = 0 \quad a_2 = 1 \quad a_3 = \phi(a_1, a_2)\]

\[a_1 = 0 \quad a_2 = 1 \quad a_3 = a_2\]

\[a_1 = 0 \quad a_3 = a_1\]

\[\phi(a_1, a_2)\]
Clean-up

- Later phases (in the code generator or separate optimization) deals with extra copies
  - Copy propagation
  - Register allocation

- Some subtle issues remain
Warning: do not “drop” version number

- You cannot just drop the version
- SSA format may have implicitly reordered operations

\[
\begin{align*}
  x &= a + b & x_1 &= a_0 + b_3 & x &= a + b \\
  y &= a + b & y_1 &= x_1 & y &= x \\
  x &= 42 & x_2 &= 42 & x &= 42 \\
  z &= a + b & z_1 &= x_1 & z &= x
\end{align*}
\]

- If we drop the version number than the program is incorrect
  - ... even if no common sub-expressions have been found
A similar problem may occur with a loop

```
a = 2;
while (...) {
    x = a;
    a = 42;
    y = a + y;
}
```

![Diagram of the loop with variables and their relationships]
A similar problem may occur with a loop

```c
a = 2;
while (...) {
    x = a;
    a = 42;
    y = a + y;
}
```

- Statement 2 \((a_2=42)\) is loop-invariant
Copies are necessary

- But how should we insert the copy?

\[
a_4 =
\]

\[
a_3 = \phi(a_2, a_1)
\]

\[
a_2 =
\]

\[
a_5 = \phi(a_4, a_2)
\]

\[
= a_5
\]
A basic block with 2 successors

- ... and there is a $\phi$ function in each successor

\[ a_3 = \phi(a_2, a_1) = a_3 \]
\[ a_2 = \]

\[ a_5 = \phi(a_4, a_2) = a_5 \]
Two copies are necessary

- 2nd copy needed only during last iteration
Copies can be problematic

- Adding an extra copy to a loop body not a good idea
  - Program likely to slow down more than it can be improved by optimization
- We can’t move the copy to the successor blocks
Copies in successor blocks

- Won’t work if successor has multiple predecessors
- Problem related to *edges*
Critical edges

- Given two basic blocks A, B in CFG, edge E: A → B.
- Edge E is a *critical edge* if A has multiple successors and B has multiple predecessors.
\[ a_4 = \quad A \quad a_2 = \]

\[ B \quad a_5 = \phi(a_4, a_2) \quad = a_5 \]
\[ a_4 = \]

\[ a_3 = \phi(a_2, a_1) \]
\[ = a_3 \]
\[ a_2 = \]

\[ a_5 = \phi(a_4, a_2) \]
\[ = a_5 \]
- Cannot insert into A
  - Has multiple successors
- Cannot insert into B
  - Has multiple predecessors

- Solution: break E and insert a new basic block
\[ a_4 = \]

\[ a_5 = \phi(a_4, a_2) = a_5 \]
\[ a_4 = \phi(a_2, a_1) = a_3 \]

\[ a_5 = \phi(a_4, a_2) = a_5 \]
Splitting an edge

- Splitting a critical edge (and inserting a basic block) may cause performance problems
  - May need extra branch/jump instruction
  - May pollute the cache
  - May not be able to optimize instructions in inserted basic block as there are only a handful instructions in the block
    - (resp. may prevent dynamic reordering of instructions)

- May be necessary for correctness
5.0 SSA-based optimizations

- Brief review
5.1 Common sub-expression elimination

- SSA makes it easy to identify local and global common sub-expressions
- Same version(s) of variables: must compute same value
- Must find places where the same expression is used (evaluated)
Value numbering

\[ x_1 = a_0 + b_3 \]
\[ y_1 = a_0 + c_2 \]

\[ \ldots \ldots \]
\[ z_1 = a_0 + b_3 \]

- **Map expressions** (with two operands) **into integers**
  - Hash function
  - Could use > 2 operands but diminishing return
Value numbering

\[
x_1 = a_0 + b_3
\]
\[
y_1 = a_0 + c_2
\]
\[
z_1 = a_0 + b_3
\]

- Map expressions (with two operands) into integers
  - Hash function
  - Could use > 2 operands but diminishing return
5.2 Constant folding

- Compiler knows where a variable is set
- Check for expressions if both operands are constant
  - If yes, evaluate in compiler
  - Usual caveats (floating point, different int representations, ....)
5.3 Removal of check for null references

- (Some) Object-oriented languages demand that compiler/runtime system check that a reference \( R \neq \text{NULL} \) before a field \( R.f \) is accessed
  - Or a method \( R.\text{foo}() \) is invoked

```
ref_1 = new Object();
...
    = ref_1.\text{foo}();
...
ref_2 = \phi(ref_1, \text{ref}_0)
...
    = ref_2.\text{foo}();
```
- SSA identifies place a variable is set
- “Normal” assignment
  - Can remove null check if RHS contains new operator
  - Can remove null check if RHS contains non-null operands

```java
ref_1 = new Object();
...
    = ref_1.foo();
...
ref_2 = φ(ref_1, ref_0)
...
    = ref_2.foo()
```
- SSA identifies place a variable is set
- “Normal” assignment
  - Can remove null check if RHS contains new operator
  - Can remove null check if RHS contains non-null operands
- \( \phi \) function
  - Simple: stop
    - i.e., insert null-check
  - Sophisticated
    - Inspect all predecessors
    - Remove null-check if defined along all paths

```java
ref_1 = new Object();
...
= ref_1.foo();
...
ref_2 = \phi(ref_1, ref_0)
...
= ref_2.foo()
```
5.4 Potentially un-initialized variables

- Assume a default assignment in the START node
  - Sets $X_0$ for all variables $X$
- If an expression uses $X_0$ as operand (or it appears as a parameter of a $\phi$ function): potentially uninitialized operand.