SSA and Psi-SSA Representations

François de Ferrière, Christophe Guillon

Francois.de-ferriere@st.com, christophe.guillon@st.com
• SSA form presentation outline:
  – SSA form definition
  – SSA form properties
  – SSA construction
  – SSA destruction
  – Optimisations over SSA form programs
SSA Form: Static Single Assignment

- Every variable as exactly one static definition

\[
\begin{align*}
x &= 2 \\
y &= x + 1 \\
x &= 3 \\
z &= x + 2
\end{align*}
\]

\[
\begin{align*}
\text{not SSA} \\
x_1 &= 2 \\
y &= x_1 + 1 \\
x_2 &= 3 \\
z &= x_2 + 2 \\
\text{SSA}
\end{align*}
\]

- It is a property of the program, not a new IR
- Motivation
  - Identify variable name and defining operation
  - Single reaching definition made explicit
SSA Property: Single Definition

- Each assignment to a variable is given a unique name (at most one definition):
  - Simple renaming for straight-line code
  - $\Phi$–nodes are introduced on control-flow join points

```
x = 1
y = x + a
y = 0
i = x + y
s = 0
Loop:
s = s + x
i = i - 1
endl:
y = y + s
```

```
x_1 = 1
y_1 = x_1 + a
y_1 = 0
x_3 = \Phi(x_1, x_2)
y_3 = \Phi(y_1, y_2)
i_1 = x_3 + y_3
s_1 = 0
Loop:
s_2 = \Phi(s_1, s_3)
i_2 = \Phi(i_1, i_3)
s_3 = s_2 + x_3
i_3 = i_2 - 1
endl:
y_4 = y_3 + s_3
```
• Each use has a reaching definition (at least one definition):
  – KILL nodes inserted to enforce definition

```
x = ...
x = ...
```

Not SSA

```
x1 = ...
x2 = ...
x3 = KILL
x4 = Φ(x2, x1, x3)
```

SSA
SSA Properties

- More compact representation than def-use chains
- Information on a variable is true everywhere (independent from the Control-Flow Graph)
- Every variable name as a known value
- Explicit merging of values
- Easy to follow use-def links, $O(1)$ time and size
- Easy to maintain def-uses chains
- Easy to rename a definition (move elimination)
- Difficult to add new variables, and thus new definitions: a new complete SSA construct pass must be performed for these new variables.
Program Intermediate Representations

• Several kind and level of Intermediate Representation (IR)
• Program IR can have SSA form property, thanks to:
  – Φ–nodes for control flow merges
  – KILL-nodes for enforcing reaching definition
  – Special nodes in case of predicated definitions (ref to PSI-SSA)
  – Additional info to track pre-allocated variables (ref to out of-SSA)
  – Other for new IR…

• Alternatives when not easy for a given IR:
  – Consider just a subset of variables
    • For instance: do not consider pre-allocated variables
  – Consider just a sub region of the program
    • For instance: SSA for basic block only
There are multiple solutions to transform a program IR into SSA form. There are common ways.

The set of nodes that need $\Phi$–nodes for any variable $V$ is the iterated dominance frontier $DF_+(L)$, where $L$ is the set of nodes with assignments to $V$.

Semi-Pruned SSA: No $\Phi$–nodes for local variables (There is always a def before a use in a basic block).

Pruned SSA: Uses live-analysis to insert $\Phi$–nodes only where they are live.
SSA Construction

• For each variable V in the program
  – Find nodes where V is defined
  – Compute the iterated dominance frontier of these nodes
  – Place Φ–nodes on the iterated dominance frontier

• Rename the variable
  – Walk the dominator tree in preorder
  – Maintain a stack of renaming for each variable
  – Create new names on definitions, rename uses
  – Fill Φ–nodes arguments in successor nodes
Φ-insertion

\[ x = 1 \quad x = a \]
\[ y = x + a \quad y = 0 \]
\[ x = \Phi(x, x) \]
\[ y = \Phi(y, y) \]
\[ i = x + y \]
\[ s = 0 \]

Loop:
\[ s = \Phi(s, s) \]
\[ i = \Phi(i, i) \]
\[ s = s + x \]
\[ i = i - 1 \]
endl:
\[ y = y + s \]

renaming

\[ x_1 = 1 \]
\[ x_2 = a \]
\[ y_1 = x_1 + a \]
\[ Y_2 = 0 \]

\[ x_3 = \Phi(x_1, x_2) \]
\[ y_3 = \Phi(y_1, y_2) \]
\[ i_1 = x_3 + y_3 \]
\[ s_1 = 0 \]

Loop:
\[ s_2 = \Phi(s_1, s_3) \]
\[ i_2 = \Phi(i_1, i_3) \]
\[ s_3 = s_2 + x_3 \]
\[ i_3 = i_2 - 1 \]
endl:
\[ y_4 = y_3 + s_3 \]
SSA Destruction: out of-SSA problem

• Issues:
  – Φ-nodes (pseudo ops) are not executable
  – pre-allocation constraints (pseudo args) must be explicited

• Out of-SSA problem:
  get a functionally equivalent program without pseudo ops or pseudo args
• Out of-SSA example:
  – From the original C code we get the IR in SSA form
  – Some transformations have been performed
  – Out-of SSA transforms the SSA form program into the executable form.

```c
int f(int a, int b)
int x=a+b;
if (x>0)
{
 a=b;
}
x=g(a)
a=x+a;
return a;
```

```
R1_0, R2_0 = pseudo_entry
x_1 = R1_0 + R2_0
b_1 = (x_1 > 0)
  \downarrow
R1_1 = R2_0
  \downarrow
R1_2 = \Phi(R1_1, R1_0)
x_2 = pseudo_call g(R1_2)
x_3 = x_2 + R1_2
R1_6_3 = x_3
pseudo_return R1_6_3
```

```
/* R1, R2 params */
x_1 = R1 + R2
b_1 = (x_1 > 0)
  \downarrow
R1 = R2
  \downarrow
R1_2 = R1
call g /* R1 = g(R1) */
x_3 = R1 + R1_2
R1 = x_3
pseudo_return R1
return /* R1 */
```
SSA Destruction: An approach

- Perform out of-SSA in two steps:
  - Convert to conventional SSA (CSSA)
  - Then perform renaming and discard pseudo ops

- Conventional SSA:
  - In this form, there is no interference between variables in a transitive closure of results and arguments of PHI operations
  - The result of the SSA construction, when no copy propagation is performed, is conventional
  - Most SSA transformations, such as copy propagation or code motion, may create non-conventional SSA
There are several algorithms to convert SSA to Conventional SSA:

- Some are wrong or do not account for IR specificities
- Some trivial algorithms insert lot of copies
- Advanced ones coalesce these copies or avoid their insertion

Alternative: maintain CSSA form all along the IR

- Very hard and bug prone: do not rely on this
Optimizations over SSA form programs

• Most standard algorithms have an SSA version, usually more efficient:
  – use-def-uses chains are maintained along transformations
  – Information on each variable is valid globally
  – Dominance property simplify algorithmic complexity

• Examples:
  – Copy Propagation: straight-forward during SSA renaming phase
  – Sparse Conditional Constant Propagation and other data flow analysis: use-def-uses chains and dominance property.
  – Dead-Code Elimination: mark side effects ops and recurse on use defs links
  – Detection of loop induction variables and determination of loop trip count is quite easy
  – Partial-redundancy elimination: still quite hard but more easy and efficient
  – Register allocation: chordal interference graph property makes coloring polynomial
The Psi-SSA form

• Psi-SSA form presentation outline:
  – Motivation
  – Definition
  – Properties
  – Benefits
  – Construction
  – Transformations
  – Destruction

• Conclusion
Motivation for Psi-SSA

- **Motivation:**
  - Enable SSA form property at machine code level

- **Why?**
  - Run all the efficient SSA based algorithms at this level (accuracy)

- **One of the issues:**
  - Target processors with full or partial support for predication
    - We can not statically determine anymore reaching definitions!

- **Predicated instructions are “optional definitions”, example:**

  \[
  \begin{align*}
  p? \ x &= 1 & \rightarrow & \text{execute } x=1 \text{ only if } p \text{ is true} \\
  !p? \ x &= 2 & \rightarrow & \text{execute } x=2 \text{ only if } !p \text{ is true}
  \end{align*}
  \]
**Definition of Psi-SSA**

- **Psi-SSA** is a SSA form (Single Static Assignment)
- **Psi-SSA** adds support for predicated instructions
  - introduce a new $\psi$ pseudo-op to keep SSA property

```plaintext
if (p)
    a_1 = 1
else
    a_2 = -1
a_3 = \phi(a_1, a_2)
if (q)
    b_1 = 0
b_2 = \phi(a_3, b_1)
b_3 = a_3 + b_2
```

**SSA representation**

```plaintext
p? a_1 = 1
!p? a_2 = -1
a_3 = \psi(p?a_1, !p?a_2)
b_1 = 0
b_2 = \psi(a_3, q?b_1)
b_3 = a_3 + b_2
```

**Psi-SSA representation**
Properties of Psi-SSA

- A Psi operation merges values defined on different predicates
- Predicates on definitions are ignored
- The result of a Psi operation is a non-predicated definition
- The execution of a Psi operation returns the value of the rightmost variable whose predicate is true at runtime

```
a_1 = 1
p? a_2 = -1
  a_3 = \psi(1?a_1,p?a_2)
q? b_1 = 0
  b_2 = \psi(1?a_1,p?a_2,q?b_1)
b_3 = a_3 + b_2
```
Properties of Psi-SSA (cont’d)

- A predicate is associated with each argument
  - Allow for speculation of predicated definitions
  - Provide support for partial predication
- Predicate domains need not be disjoint
  - Several predicates may be true at the same time.
  - The order of the arguments in a Psi operation is significant

\[
\begin{align*}
a_1 &= 1 \\
p? a_2 &= -1 \\
a_3 &= \psi(1?a_1,p?a_2) \\
q? b_1 &= 0 \\
b_2 &= \psi(1?a_1,p?a_2,q?b_1) \\
b_3 &= a_3 + b_2
\end{align*}
\]
Benefits of Psi-SSA

• Easy to implement on top of an SSA representation

• No penalty if no predicated operation

• More flexibility in optimization ordering for predicated instruction sets
  – SSA algorithms are easy to adapt to the Psi-SSA representation (just add the support for the new pseudo op)
  – If-Conversion under SSA

• Specific optimizations on predicated code
  – Predicate promotion
Benefits of Psi-SSA (cont’d)

• Standard SSA algorithms can be used on Psi-SSA
  – Predicated instructions are treated as unconditional
  – New rules have to be defined on Psi operations
  – constant propagation, dead code elimination, global value numbering have been adapted to this representation

• Example: Constant propagation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_1 = 1)</td>
<td>1</td>
</tr>
<tr>
<td>(p? a_2 = a_1+1)</td>
<td>2</td>
</tr>
<tr>
<td>(!p? a_3 = 2)</td>
<td>2</td>
</tr>
<tr>
<td>(a_4 = \psi(p?a_2, !p?a_3))</td>
<td>2</td>
</tr>
</tbody>
</table>
Construction of Psi-SSA

• During the SSA construction
  – Insertion of Psi operation after predicated definitions

\[
\begin{align*}
    a_1 &= 0 \\
    \text{if } (p) \quad &a_2 = 1 \\
    a_3 &= \psi(1?a_1, p?a_2)
\end{align*}
\]

• While in SSA form by an if-conversion algorithm
  – Transformation of Phi operations into Psi operations

\[
\begin{align*}
    a_1 &= 0 \\
    \text{if } (p) \quad &a_2 = 1 \\
    a_3 &= \Phi(a_1, a_2) \\
    p? a_2 &= 1 \\
    a_3 &= \psi(1?a_1, p?a_2)
\end{align*}
\]
Transformations on Psi-SSA

- **Some transformations on Psi operations:**
  - **Psi-inlining**
    \[
    \begin{align*}
    x &= \psi(p?a, q?b) \\
    y &= \psi(p | q?x, r?c) \\
    \rightarrow y &= \psi(p?a, q?b, r?c)
    \end{align*}
    \]
  - **Psi-reduction**
    \[
    \begin{align*}
    x &= \psi(p?a, q?b, p?c) \\
    \rightarrow x &= \psi(q?b, p?c)
    \end{align*}
    \]
  - **Psi-projection**
    \[
    \begin{align*}
    x &= \psi(p?a, q?b) \quad /* \quad p \cap q = \emptyset */ \\
    \rightarrow x_1 &= \psi(p?a) \\
    \quad p? z=x \quad /* \quad single \ use \ of \ x*/ \\
    \rightarrow p? z=x_1 \quad //z=a
    \end{align*}
    \]
  - **Psi-promotion**
    \[
    \begin{align*}
    x &= \psi(p?a, q?b) \\
    \rightarrow x &= \psi(1?a, q?b)
    \end{align*}
    \]
Destruction of Psi-SSA

- Variables connected through a Psi operation must be renamed into a single variable, but:
  - Code motion may have changed the order in which predicated definitions occur
  - Operation speculation may have assigned a different predicate on a variable’s definition and on its use in a Psi operation
  - Copy folding may have introduced interferences between variables in Psi operations

\[
\begin{array}{c}
p? a_2 = 1 \\
a_1 = 0 \\
a_3 = \psi(1? a_1, p? a_2)
\end{array}
\quad
\begin{array}{c}
a_1 = 0 \\
p? a_2 = 1 \\
a_3 = \psi(1? a_1, p? a_2)
\end{array}
\]

\[
\begin{array}{c}
a_1 = 0 \\
a_2 = 1 \\
a_3 = \psi(1? a_1, q? a_2)
\end{array}
\quad
\begin{array}{c}
a_1 = 0 \\
p? a_2 = 1 \\
a_3 = \psi(1? a_1, p? a_2)
\end{array}
\]

\[
\begin{array}{c}
a_1 = 0 \\
a_2 = 1 \\
a_3 = \psi(1? a_1, q? a_2)
\end{array}
\quad
\begin{array}{c}
a_1 = 0 \\
p? a_2 = 1 \\
a_3 = \psi(1? a_1, p? a_2)
\end{array}
\]
Destruction of Psi-SSA (cont’d)

- Implemented as two steps above the out of-SSA algorithm

- A Psi-Normalize step
  - Restores the order of predicated definitions
  - Uses the same predicate on a variable’s definition and on its use in a Psi operation

- A Psi-congruence step
  - Insert copies to remove interferences in psi-congruence classes
  - Uses a special definition for liveness on normalized Psi operations

\[
p? a_1 = \\
q? a_2 = \\
a_3 = \psi(p?a_1, q?a_2)
\]
Destruction of Psi-SSA (cont’d)

• Predicated copies are generated to repair non-normalized Psi operations and interferences between Psi arguments

• Interferences between Psi arguments must also take into account interferences on Phi operations

• A simple Predicate Query System is used to eliminate false interferences on disjoint predicates
Conclusion

- Algorithms to build, optimize and deconstruct the Psi-SSA representation are well defined.

- The Psi-SSA representation has proven to be a very effective representation to applying transformations on predicated code for our target processors.

- Standard SSA algorithms are easy to adapt to Psi-SSA.

- The Psi-SSA representation gives more flexibility in the ordering of optimizations in the compiler back-end.
• Compilers using SSA
  – Middle-end: gcc, open64
  – Machine level: open64 (at ST), LAO (at ST), llvm, HotSpot
• Our contributions to SSA/Psi-SSA representation:
  – “Optimizing Translation Out of SSA Using Renaming Constraints”
    F. de Ferrière, C.Guillon, F.Rastello – CGO-2004
  – “Revisiting Out of SSA Translation for Correctness, Efficiency and Speed”
  – “Efficient static single assignment form for predication”
    A.Stouchinin, F. de Ferrière - Micro-34
  – “Improvements to the Psi-SSA Representation”
    F. de Ferrière – Scopes 2007