An extension to the SSA representation for predicated code

April 29th, 2009

François de Ferrière
SSA For Predicated Code

• What is different with predicated code

• An extension to SSA for predicated code

• Going out-of-SSA requires additional work

• Conclusion
Why predicated code under SSA

• Internal representation is at target instruction level
• Our target processors have full or partial support for predication
• Some optimizations can generate predicated code
  – Code selection
  – Peephole transformations
  – If-conversion algorithm
• We need SSA for various optimizations
  – Value-range analysis
  – Target specific optimizations
  – If-conversion
Different levels of support for predication

- **A select instruction**
  - But this not really predicated code
    
    ```
    a = load @...  
    b = add ...  
    c = select p ? a : b
    ```

- **Only MOV instructions are predicated**
  
  ```
  a = load @...  
  b = add ...  
  p? c = a  
  !p? c = b
  ```

- **Most instructions are predicated**
  
  ```
  p? c = load @...  
  !p? c = add ...
  ```
What is different with predicated code

• A use may refer to several optional definitions:

<table>
<thead>
<tr>
<th>Non SSA form</th>
<th>SSA form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = \text{load @}\ldots$</td>
<td>$a_1 = \text{load @}\ldots$</td>
</tr>
<tr>
<td>$b = 0$</td>
<td>$b_1 = 0$</td>
</tr>
<tr>
<td>$p?\ a = 0$</td>
<td>$p?\ a_2 = 0$</td>
</tr>
<tr>
<td>$p?\ b = a$</td>
<td>$p?\ b_2 = a_2 \text{ or } a_1$</td>
</tr>
<tr>
<td>$!p?\ a = b$</td>
<td>$!p?\ a_3 = b_2 \text{ or } b_1$</td>
</tr>
</tbody>
</table>

– When definitions are renamed, how to rename uses?
What is different with predicated code (Cont’d)

- First solution: no renaming of predicated definitions
  - Variables defined on predicated operations are not renamed into SSA variables

\[
\begin{array}{l}
a = \text{load @}...
b = 0 
p? a = 0 
p? b = a 
!p? a = b 
\end{array}
\]

- This may have a large impact even if predication is used on a few instructions
• Second solution: Add an implicit use on predicated instructions

\[
\begin{align*}
a_1 &= \text{load } @... \\
b_1 &= 0 \\
p? a_2 &= 0[,,a_1] \\
p? b_2 &= a_2[,,b_1] \\
!p? a_3 &= b_2[,,a_2]
\end{align*}
\]

– Non-predicated definitions/uses can still benefit from the SSA form
– This is a significant modification in the intermediate representation
– Predicated code is still difficult to analyze/optimize
Third solution: A `select` instruction is used to express the semantics of a predicated definition

```plaintext
a_1 = load @...
b_1 = 0
p? a_2 = 0
  a_3 = select p ? a_2 : a_1
p? b_2 = a_3
  b_3 = select p ? b_2 : b_1
!p? a_4 = b_3
  a_5 = select !p ? a_4 : a_3
```

- Only one instruction is added in the intermediate representation
- Peephole optimizations on the `select` instruction can be used to optimize predicated code
An extension to SSA for predicated code

• A new pseudo instruction : \( \psi \)

\[
\begin{align*}
\text{p? } a_1 & = \text{load } @... \\
\text{!p? } a_2 & = \text{add } ... \\
a_3 & = \text{select } p \ ? \ a_1 : a_2 \\
& = a_3
\end{align*}
\]

\[
\begin{align*}
\text{p? } a_1 & = \text{load } @... \\
\text{!p? } a_2 & = \text{add } ... \\
a_3 & = \psi(p?a_1, !p?a_2) \\
& = a_3
\end{align*}
\]

• Generalization of the semantics of a \texttt{select} instruction
  – 1, 2 or more arguments
  – Each argument has an associated predicate
  – The result is the value of the rightmost argument whose predicate is TRUE at execution time.
  – The predicates need not be disjoint
  – The order of the arguments is significant

• A predicated definition can be used in several \( \psi \) operations
This is still standard SSA

• A $\Psi$ instruction is a regular instruction
  – It is not different from any other instructions in the intermediate representation
  – It has a simple semantics, without side effects
  – There is no restriction on the variable defined on a $\Psi$ instruction, in particular it can be used in $\Phi$ operations

• Predicated definitions are now real definitions
  – For SSA analysis and optimizations, a variable defined on a predicated operation is an unconditional definition
  – Predicated instructions can be moved with the same rules as non-predicated ones

• By construction, uses of a predicated definition will only occur in $\Psi$ instructions
Predicated code can easily be optimized

- Local analysis and transformations on $\psi$ operations are enough to optimize predicated code

\[
\begin{align*}
    a_1 &= \text{load @...} \\
    b_1 &= 0 \\
    p? a_2 &= 0 \\
    a_3 &= \psi(1?a_1, p?a_2) \\
    p? b_2 &= a_3 \\
    b_3 &= \psi(1?b_1, p?b_2) \\
    !p? a_4 &= b_3 \\
    a_5 &= \psi(1?a_1, p?a_2, !p?a_4)
\end{align*}
\]
Predicated code can easily be optimized

• Local analysis and transformations on $\psi$ operations are enough to optimize predicated code

\[
\begin{align*}
a_1 &= \text{load @...} \\
b_1 &= 0 \\
p? a_2 &= 0 \\
a_3 &= \psi(1?a_1, p?a_2) \\
a_6 &= \psi(p?a_2) \\
p? b_2 &= a_3 \\
b_3 &= \psi(1?b_1, p?b_2) \\
b_4 &= \psi(!p?b_1) \\
!p? a_4 &= b_3 \\
a_5 &= \psi(1?a_1, p?a_2, !p?a_4) \\
a_7 &= \psi(p?a_2, !p?a_4)
\end{align*}
\]
Predicated code can easily be optimized

- Local analysis and transformations on $\psi$ operations are enough to optimize predicated code

\[
\begin{align*}
    a_1 &= \text{load} @... \\
    b_1 &= 0 \\
    \text{p?} \ a_2 &= 0 \\
    a_3 &= \psi(1?a_1, p?a_2) \\
    a_6 &= \psi(p?a_2) \\
    \text{p?} \ b_2 &= a_3 \\
    b_3 &= \psi(1?b_1, p?b_2) \\
    b_4 &= \psi(!p?b_1) \\
    \text{!p?} \ a_4 &= b_3 \\
    a_5 &= \psi(1?a_1, p?a_2, !p?a_4) \\
    a_7 &= \psi(p?a_2, !p?a_4) \\
    \text{b}_4 &= 0 \\
    a_7 &= 0
\end{align*}
\]
• When going out of SSA, $\psi$ operations are similar to $\Phi$ operations

• Simple elimination
  – A $\psi$ operation can be replaced by predicated copies for each of its arguments.
  – But the resulting predicated copies will not be easily coalesced

• Optimized elimination
  – Interferences between arguments in $\psi$ operations are analyzed
  – A predicate query system is used to eliminate false interferences between definitions on disjoint predicates
Going out of SSA requires additional work (Cont’d)

• Needs to restore the semantics of the Psi for non-disjoint predicates
  • The order of the definitions may have to be repaired
  • Speculation may require predicated copies

\[ \begin{align*}
!p? a_2 &= \text{add } \ldots \\
a_1 &= \text{load } @\ldots \\
a_3 &= \psi(p?a_1,!p?a_2) \\
&= a_3
\end{align*} \]

\[ \begin{align*}
!p? a_2 &= \text{add } \ldots \\
a_1 &= \text{load } @\ldots \\
!p? a_4 &= a_2 \\
a_3 &= \psi(1?a_1,!p?a_2) \\
&= a_3
\end{align*} \]

• Then, the elimination of the Psi is a coalescing problem
  • Similar to coalescing on Phi operations
  • Done at the same time as elimination of PHI
Conclusion

- This SSA extension for predicated code is easy to implement on top of an SSA representation.
- There is no penalty if no predicated operation.
- It gives more flexibility in optimization ordering:
  - Optimizations that generate predicated code can be performed before going in SSA or directly on the SSA representation.
- Standard SSA algorithms are easy to adapt to this SSA extension.
- Optimization of predicated code is simple under this representation.
More on SSA for predicated code

- Two publications describe the Psi-SSA representation:
  - “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