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 :

```
    a = load @...
    b = 0
 p? a = 0
 p? b = a
!p? a = b
```

Non SSA form

$$
\begin{array}{l}
\quad a_1 = \texttt{load @...} \\
\quad b_1 = 0 \\
p?\ a_2 = 0 \\
p?\ b_2 = a_2 \text{ or } a_1 \\
!p?\ a_3 = b_2 \text{ or } b_1
\end{array}
$$

SSA form

  – 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

```
    a = load @...
    b = 0
 p? a = 0
 p? b = a
!p? a = b
```

  - This may have a large impact even if predication is used on a few instructions

What is different with predicated code (Cont'd)

- Second solution: Add an implicit use on predicated instructions

```
    a1 = load @...
    b1 = 0
 p? a2 = 0[,a1]
 p? b2 = a2[,b1]
!p? a3 = b2[,a2]
```

  - 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

```
    a1 = load @...
    b1 = 0
 p? a2 = 0
    a3 = select p ? a2 : a1
 p? b2 = a3
    b3 = select p ? b2 : b1
!p? a4 = b3
    a5 = select !p ? a4 : a3
```

  - 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 : ψ

```
 p? a1 = load @...
!p? a2 = add ...
    a3 = select p ? a1 : a2
       = a3
```

```
 p? a1 = load @...
!p? a2 = add ...
    a3 = ψ(p?a1,!p?a2)
       = a3
```

- Generalization of the semantics of a `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 ψ operations

This is still standard SSA

- A ψ 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 ψ instruction, in particular it can be used in Φ 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 ψ instructions

Predicated code can easily be optimized

- Local analysis and transformations on ψ operations are enough to optimize predicated code

```
    a1 = load @...
    b1 = 0
 p? a2 = 0
    a3 = ψ(1?a1,p?a2)

 p? b2 = a3
    b3 = ψ(1?b1,p?b2)

!p? a4 = b3
    a5 = ψ(1?a1,p?a2,!p?a4)
```

Predicated code can easily be optimized

- Local analysis and transformations on ψ operations are enough to optimize predicated code

```
     a1 = load @...
     b1 = 0
 p?  a2 = 0
     a3 = ψ(1?a1,p?a2)
     a6 = ψ(p?a2)
 p?  b2 = a3
     b3 = ψ(1?b1,p?b2)
     b4 = ψ(!p?b1)
!p?  a4 = b3
     a5 = ψ(1?a1,p?a2,!p?a4)
     a7 = ψ(p?a2,!p?a4)
```

⇒

```
     b1 = 0
 p?  a2 = 0
     a6 = ψ(p?a2)
 p?  b2 = a6
     b4 = ψ(!p?b1)
!p?  a4 = b4
     a7 = ψ(p?a2,!p?a4)
```

Predicated code can easily be optimized

- Local analysis and transformations on ψ operations are enough to optimize predicated code

```
    a1 = load @...
    b1 = 0
 p? a2 = 0
    a3 = ψ(1?a1,p?a2)
    a6 = ψ(p?a2)
 p? b2 = a3
    b3 = ψ(1?b1,p?b2)
    b4 = ψ(!p?b1)
!p? a4 = b3
    a5 = ψ(1?a1,p?a2,!p?a4)
    a7 = ψ(p?a2,!p?a4)
```

⇒

```
    b1 = 0
 p? a2 = 0
    a6 = ψ(p?a2)
 p? b2 = a6
    b4 = ψ(!p?b1)
!p? a4 = b4
    a7 = ψ(p?a2,!p?a4)
```

⇓

```
    b4 = 0
    a7 = 0
```

Going out of SSA requires additional work

- When going out of SSA, ψ operations are similar to Φ operations
- Simple elimination
  - A ψ 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 ψ 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

```
!p? a2 = add ...
    a1 = load @...
    a3 = ψ(p?a1,!p?a2)
       = a3
```

⇒

```
!p? a2 = add ...
    a1 = load @...
!p? a4 = a2
    a3 = ψ(1?a1,!p?a2)
       = a3
```

- 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 generates 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