Topic I (d):

Static Single Assignment Form (SSA)
Reading List

- Slides: Topic Ix
- Other readings as assigned in class
ABET Outcome

- Ability to apply knowledge of SSA technique in compiler optimization
- An ability to formulate and solve the basic SSA construction problem based on the techniques introduced in class.
- Ability to analyze the basic algorithms using SSA form to express and formulate dataflow analysis problem
- A Knowledge on contemporary issues on this topic.
Roadmap

- Motivation
- Introduction:
  - SSA form
  - Construction Method
  - Application of SSA to Dataflow Analysis Problems
- PRE (Partial Redundancy Elimination) and SSAPRE
- Summary
Prelude

(SSA: A program is said to be in SSA form iff
- Each variable is statically defined exactly only once, and
- each use of a variable is dominated by that variable’s definition.

So, is straight line code in SSA form ?

Example

In general, how to transform an arbitrary program into SSA form?

Does the definition of $X_2$ dominate its use in the example?
SSA: Motivation

- Provide a uniform basis of an IR to solve a wide range of classical dataflow problems
- Encode both dataflow and control flow information
- A SSA form can be constructed and maintained efficiently
- Many SSA dataflow analysis algorithms are more efficient (have lower complexity) than their CFG counterparts.
### Algorithm Complexity

Assume a 1 GHz machine, and an algorithm that takes \( f(n) \) steps (1 step = 1 nanosecond).

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>128</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>8 ns</td>
<td>16 ns</td>
<td>32 ns</td>
<td>128 ns</td>
<td>1024 ns</td>
</tr>
<tr>
<td>( \lg n )</td>
<td>3 ns</td>
<td>4 ns</td>
<td>5 ns</td>
<td>7 ns</td>
<td>10 ns</td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>2.8 ns</td>
<td>4 ns</td>
<td>6 ns</td>
<td>11 ns</td>
<td>32 ns</td>
</tr>
<tr>
<td>( n \sqrt{n} )</td>
<td>8 ns</td>
<td>16 ns</td>
<td>32 ns</td>
<td>128 ns</td>
<td>1 ( \mu )s</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>24 ns</td>
<td>64 ns</td>
<td>160 ns</td>
<td>896 ns</td>
<td>10 ( \mu )s</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>64 ns</td>
<td>256 ns</td>
<td>1.0 ( \mu )s</td>
<td>16 ( \mu )s</td>
<td>1 ms</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>512 ns</td>
<td>4 ( \mu )s</td>
<td>32.8 ( \mu )s</td>
<td>2 ms</td>
<td>1.1 sec</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>256 ns</td>
<td>66 ( \mu )s</td>
<td>4 sec.</td>
<td>10^{22} year</td>
<td></td>
</tr>
<tr>
<td>( n! )</td>
<td>40 ( \mu )s</td>
<td>5.8 hours</td>
<td>10^{19} year</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Where SSA Is Used In Modern Compilers?

Good IR

Front end

Interprocedural Analysis and Optimization

Loop Nest Optimization and Parallelization

Global (Scalar) Optimization

Middle-End

Backend

Code Generation
KCC Compiler Infrastructure

Front end

Fortran

C

C++

fe90

gfec

gfecc

Middle end

Machine Model

Very High WHIRL

High WHIRL

Middle WHIRL

Low WHIRL

Very Low WHIRL

CGIR

Back end

Asm file

• Source to IR (Scanner → Parser → RTL → WHIRL)

• VHO (Very High WHIRL Optimizer)
  • Standalone Inliner
  • W2C/W2F

• IPA (inter-procedural analysis & opt)
  • LNO (Loop unrolling/fission/fusion/tiling/peeling etc)
  • PREOPT (point-to analysis etc)

• WOPT
  - SSAPRE (Partial Redundancy Elimination)
  - VNFRE (Value numbering based full redundancy elim.)
  • RVI-1 (Register Variable Identification)

• RVI-2

• Some peephole opt

• Cflow (control flow opt)
  • EBO (extended block opt.)
  • PQS (predicate query system)
  • Loop Opt (Unrolling + SWP)
  • GCM (global code motion), HB sched (hyperblk schedule)
  • GRA/LRA (global/local register alloc)
Roadmap

- Motivation
- Introduction:
  - SSA form
  - Construction Method
  - Application of SSA to Dataflow Analysis Problems
- PRE (Partial Redundancy Elimination) and SSAPRE
- Summary
Each variable has only one definition in the program text.

This single static definition can be in a loop and may be executed many times. Thus even in a program expressed in SSA, a variable can be dynamically defined many times.
Advantages of SSA

- Simpler dataflow analysis
- No need to use use-def/def-use chains, which requires $N \times M$ space for $N$ uses and $M$ definitions
- SSA form relates in a useful way with dominance structures.
SSA Form – An Example

SSA-form
- Each name is defined exactly once
- Each use refers to exactly one name

What’s hard
- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form
- Insert Ø-functions at birth points
- Rename all values for uniqueness

*Curtesy: Slide 10-14 are from the book from Prof. K. Cooper’s website*
Birth Points

*(another notion due to Tarjan)*

Consider the flow of values in this example:

- The value x appears everywhere.
- It takes on several values.
- Here, x can be 13, y-z, or 17-4.
- Here, it can also be a+b.

If each value has its own name ...:
- Need a way to merge these distinct values.
- Values are “born” at merge points.
Birth Points
(another notion due to Tarjan)

Consider the flow of values in this example:

\[
\begin{align*}
x &\leftarrow 17 - 4 \\
x &\leftarrow a + b \\
x &\leftarrow y - z \\
x &\leftarrow 13 \\
z &\leftarrow x \times q \\
s &\leftarrow w - x
\end{align*}
\]

New value for \(x\) here:
- 17 - 4 or \(y - z\)
- 13 or \((17 - 4 \text{ or } y - z)\)
- \(a + b\) or \(((13 \text{ or } (17 - 4 \text{ or } y - z))\)
Birth Points
(another notion due to Tarjan)

Consider the value flow below:

\[x \leftarrow 17 - 4\]

\[x \leftarrow a + b\]

\[x \leftarrow y - z\]

\[x \leftarrow 13\]

\[z \leftarrow x \times q\]

\[s \leftarrow w - x\]

- All birth points are join points
- Not all join points are birth points
- Birth points are value-specific ...

These are all birth points for values
Review

SSA-form
- Each name is defined exactly once
- Each use refers to exactly one name

What’s hard
- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form
- Insert $\emptyset$-functions at birth points
- Rename all values for uniqueness

A $\emptyset$-function is a special kind of copy that selects one of its parameters.
The choice of parameter is governed by the CFG edge along which control reached the current block.

Real machines do not implement a $\emptyset$-function directly in hardware. (not yet!)
Use-def Dependencies in Non-straight-line Code

Many uses to many defs
- Overhead in representation
- Hard to manage
Factoring Operator $\phi$

Factoring – when multiple edges cross a join point, create a common node $\Phi$ that all edges must pass through

- Number of edges reduced from 9 to 6
- $A \phi$ is regarded as def (its parameters are uses)
- Many uses to 1 def
- Each def dominates all its uses
Rename to represent use-def edges

- No longer necessary to represent the use-def edges explicitly
Is this code in SSA form?

No, two definitions of $a$ at B4 appear in the code (in B1 and B3)

How can we transform this code into a code in SSA form?

We can create two versions of $a$, one for B1 and another for B3.
SSA Form in Control-Flow Path Merges

But which version should we use in B4 now?

We define a *fictional* function that “knows” which control path was taken to reach the basic block B4:

\[
\phi(a_1, a_2) = \begin{cases} 
  a_1 & \text{if we arrive at B4 from B2} \\
  a_2 & \text{if we arrive at B4 from B3} 
\end{cases}
\]
But, which version should we use in B4 now?

We define a fictional function that “knows” which control path was taken to reach the basic block B4:

\[ \phi(a_2, a_1) = \begin{cases} 
    a_1 & \text{if we arrive at B4 from B2} \\
    a_2 & \text{if we arrive at B4 from B3} 
\end{cases} \]
A Loop Example

\[ a \leftarrow 0 \]
\[ b \leftarrow a + 1 \]
\[ c \leftarrow c + b \]
\[ a \leftarrow b \times 2 \]
if \( a < N \)
return

\[ a1 \leftarrow 0 \]
\[ a3 \leftarrow \phi(a1, a2) \]
\[ b2 \leftarrow \phi(b0, b2) \]
\[ c2 \leftarrow \phi(c0, c1) \]
\[ b2 \leftarrow a3 + 1 \]
\[ c1 \leftarrow c2 + b2 \]
\[ a2 \leftarrow b2 \times 2 \]
if \( a2 < N \)
return

\( \phi(b0, b2) \) is not necessary because \( b0 \) is never used. But the phase that generates \( \phi \) functions does not know it. Unnecessary functions are eliminated by dead code elimination.

**Note:** only \( a, c \) are first used in the loop body before it is redefined. For \( b \), it is redefined right at the beginning!
The $\phi$ function

How can we implement a $\phi$ function that “knows” which control path was taken?

Answer 1: We don’t!! The $\phi$ function is used only to connect use to definitions during optimization, but is never implemented.

Answer 2: If we must execute the $\phi$ function, we can implement it by inserting MOVE instructions in all control paths.
Roadmap

• Motivation

• Introduction:
  ▶ SSA form
  ▶ Construction Method
  ▶ Application of SSA to Dataflow Analysis Problems

• PRE (Partial Redundancy Elimination) and SSAPRE

• Summary
Criteria For Inserting $\phi$ Functions

We could insert one $\phi$ function for each variable at every *join* point (a point in the CFG with more than one predecessor). But that would be wasteful.

What should be our criteria to insert a $\phi$ function for a variable $a$ at node $z$ of the CFG?

Intuitively, we should add a function $\phi$ if there are two definitions of $a$ that can reach the point $z$ through distinct paths.
A naïve method

- Simply introduce a $\phi$-function at each “join” point in CFG
- But, we already pointed out that this is inefficient – too many useless $\phi$-functions may be introduced!
- What is a good algorithm to introduce only the right number of $\phi$-functions?
Path Convergence Criterion

Insert a $\phi$ function for a variable $a$ at a node $z$ if all the following conditions are true:
1. There is a block $x$ that defines $a$
2. There is a block $y \neq x$ that defines $a$
3. There is a non-empty path $P_{xz}$ from $x$ to $z$
4. There is a non-empty path $P_{yz}$ from $y$ to $z$
5. Paths $P_{xz}$ and $P_{yz}$ don’t have any nodes in common other than $z$
6. ?

The start node contains an implicit definition of every variable.
Iterated Path-Convergence Criterion

The $\phi$ function itself is a definition of $a$. Therefore the path-convergence criterion is a set of equations that must be satisfied.

while there are nodes $x$, $y$, $z$ satisfying conditions 1-5 and $z$ does not contain a $\phi$ function for $a$
do insert $a \leftarrow \phi(a, a, ..., a)$ at node $z$

This algorithm is extremely costly, because it requires the examination of every triple of nodes $x$, $y$, $z$ and every path leading from $x$ to $y$.

Can we do better? – a topic for more discussion
Concept of dominance Frontiers

An Intuitive View

Blocks dominate by bb1

Border between dorm and not-dorm (Dominance Frontier)
Dominance Frontier

- The *dominance frontier* $\text{DF}(x)$ of a node $x$ is the set of all node $z$ such that $x$ dominates a predecessor of $z$, without strictly dominating $z$.

Recall: if $x$ dominates $y$ and $x \neq y$, then $x$ *strictly* dominates $y$.
Calculate The Dominance Frontier

An Intuitive Way

How to Determine the Dominance Frontier of Node 5?

1. Determine the dominance region of node 5:
   \{5, 6, 7, 8\}

2. Determine the targets of edges crossing from the dominance region of node 5
   These targets are the dominance frontier of node 5
   \(DF(5) = \{4, 5, 12, 13\}\)

NOTE: node 5 is in \(DF(5)\) in this case – why?
Are we done?

- Not yet!
- See a simple example ..
Putting program into SSA form

- \( \Phi \) needed only at dominance frontiers of defs (where it stops dominating)
- Dominance frontiers pre-computed based on control flow graph
- Two phases:
  1. Insert \( \Phi \)’s at dominance frontiers of each def (recursive)
  2. Rename the uses to their def's name
     - Maintain and update stack of variable versions in pre-order traversal of dominator tree
Example

Phase 1: $\Phi$ Insertion

Steps:
def at BB 3 $\rightarrow$ $\Phi$ at BB 4
$\Phi$ def at BB 4 $\rightarrow$ $\Phi$ at BB 2
Example

Phase 2: Rename

dominator tree

\[ a_1 = \]

stack for \( a \)

\[ a = \phi(a, a_1) \]

\[ a = \phi(a, a) \]
Example

Phase 2: Rename

dominator tree
Example

Phase 2: Rename

\[ a_1 = \]

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

\[ a_3 = \]

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

dominator tree
Example

Phase 2: Rename

dominator tree

\[ a_1 = \]

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

\[ a_3 = \]

\[ a_4 = \phi(a_2, a_3) \]
Roadmap

*z Motivation
*z Introduction:
  ▶ SSA form
  ▶ Construction Method
  ▶ Application of SSA to Dataflow Analysis Problems
*z PRE (Partial Redundancy Elimination) and SSAPRE
*z Summary
If there is a statement \( v \leftarrow c \), where \( c \) is a constant, then all uses of \( v \) can be replaced for \( c \).

A \( \phi \) function of the form \( v \leftarrow \phi(c_1, c_2, \ldots, c_n) \) where all \( c_i \)'s are identical can be replaced for \( v \leftarrow c \).

Using a work list algorithm in a program in SSA form, we can perform constant propagation in linear time.

In the next slide we assume that \( x, y, z \) are variables and \( a, b, c \) are constants.
Linear Time Optimizations in SSA form

**Copy propagation**: The statement $x \leftarrow \phi(y)$ or the statement $x \leftarrow y$ can be deleted and $y$ can substitute every use of $x$.

**Constant folding**: If we have the statement $x \leftarrow a \oplus b$, we can evaluate $c \leftarrow a \oplus b$ at compile time and replace the statement for $x \leftarrow c$.

**Constant conditions**: The conditional

if $a < b$ goto L1 else L2

can be replaced for goto L1 or goto L2, according to the compile time evaluation of $a < b$, and the CFG, use lists, adjust accordingly.

**Unreachable Code**: eliminate unreachable blocks.
Dead-Code Elimination in SSA Form

Because there is only one definition for each variable, if the list of uses of the variable is empty, the definition is dead.

When a statement $v \leftarrow x \oplus y$ is eliminated because $v$ is dead, this statement should be removed from the list of uses of $x$ and $y$. Which might cause those definitions to become dead.

Thus we need to iterate the dead code elimination algorithm.
A Case Study: Dead Store Elimination

Steps:
1. Assume all defs are dead and all statements not required
2. Mark following statements required:
   a. Function return values
   b. Statements with side effects
   c. Def of global variables
3. Variables in required statements are live
4. Propagate liveness backwards iteratively through:
   a. use-def edges – when a variable is live, its def statement is made live
   b. control dependences
Control Dependence

- Statements in branched-to blocks depend on the conditional branch
- Equivalent to post-dominance frontier (dominance frontier of the inverted control flow graph)
Example of dead store elimination

Propagation steps:

1. \( \text{return } s_2 \rightarrow s_2 \)
2. \( s_2 \rightarrow s_2 = s_3 \times s_3 \)
3. \( s_3 \rightarrow s_3 = \phi(s_2, s_1) \)
4. \( s_1 \rightarrow s_1 = \)
5. \( \text{return } s_2 \rightarrow \text{if } (i_2 < 10) \)
   [control dependence]
6. \( i_2 \rightarrow i_2 = i_3 + 1 \)
7. \( i_3 \rightarrow i_3 = \phi(i_2, i_1) \)
8. \( i_1 \rightarrow i_1 = \)

Nothing is dead
Example of dead store elimination
All statements not required; whole loop deleted

\[ i_3 = \phi(i_2, i_1) \]
\[ s_3 = \phi(s_2, s_1) \]
\[ i_2 = i_3 + 1 \]
\[ s_2 = s_3 \times s_3 \]
\[ \text{if } (i_3 < 10) \]

\[ i_1 = \]
\[ s_1 = \]

\[ \text{empty} \]
Advantages of SSA-based optimizations

Dependency information built-in
- No separate phase required to compute dependency information

Transformed output preserves SSA form
- Little overhead to update dependencies

Efficient algorithms due to:
- Sparse occurrence of nodes
  - Complexity dependent only on problem size (independent of program size)
- Linear data flow propagation along use-def edges
- Can customize treatment according to candidate

Can re-apply algorithms as often as needed

No separation of local optimizations from global optimizations