Static Single Assignment Form

Kenneth Zadeck

NaturalBridge, Inc.
zadeck@naturalbridge.com

NaturalBridge

- NaturalBridge is a software development and consulting company.
- We built a Java VM with a static compiler.
  - 100% SSA based static compiler.
- NaturalBridge has been retained by Apple to aid in the SSA development of GCC.

My Role

- Assess the SSA algorithms in GCC.
  - Choice of algorithm
  - Quality of implementation
  - Integration issues
- Implement changes to improve performance of the compiler and the generated code.
- Assist people with SSA or other algorithm issues.

History of Static Single Assignment SSA

- Really invented by Shapiro & Saint in 1975.
- Developed at IBM Watson Lab 1984-1990.
- Principal developers:

-Bowen Alpern

-Jeanne Ferrante

-Mark Wegman

-Ron Cytron

-Barry Rosen

-Kenneth Zadeck

What is Static Single Assignment Form

What is Static Single Assignment Form?

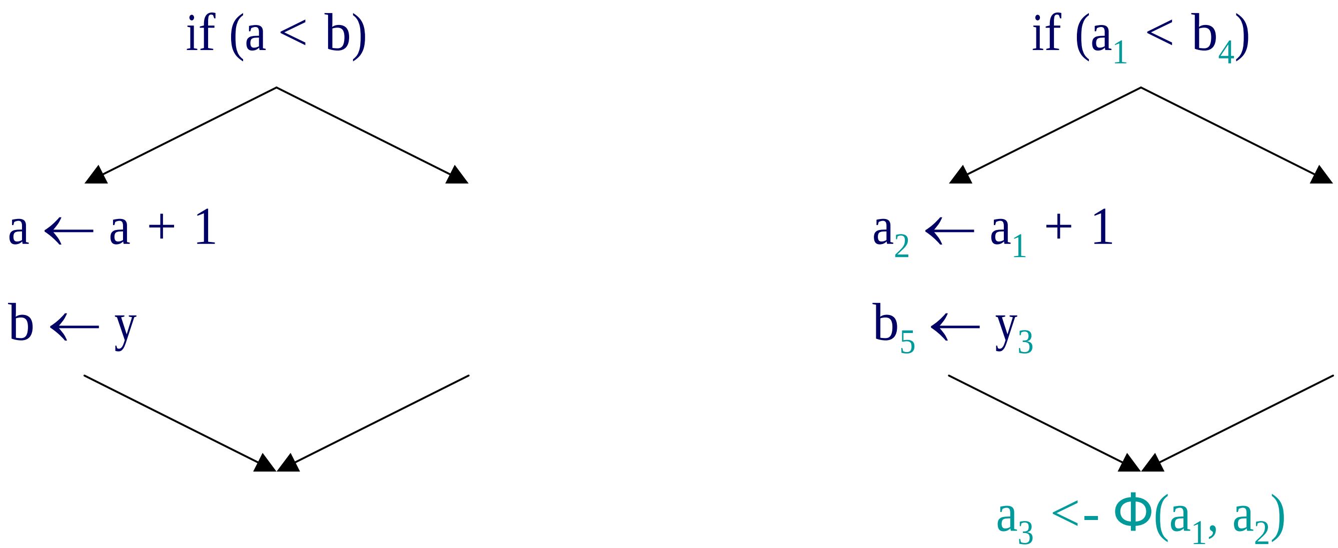
if (a < b)

$a \leftarrow a + 1$

$b \leftarrow y$

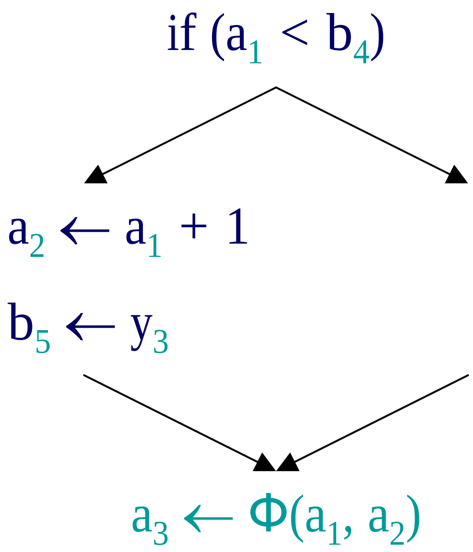
What is Static Single Assignment Form?

- A systematic splitting of the live range of a variable to remove spurious dependencies.



What is Static Single Assignment Form?

- Each assignment is to a unique variable.
  - This allows information to be associated with the value rather than the variable.
  - Assignments can move to places where many values are simultaneously live.
- Each use is reached from exactly one assignment.
- Φ-functions are inserted where values are joined.



What is Static Single Assignment Form?

- An inexpensive way to get better information out of flow insensitive analysis algorithms.
- A way of representing finer grained variable specific information.
- A way of decreasing the size of def-use chains.

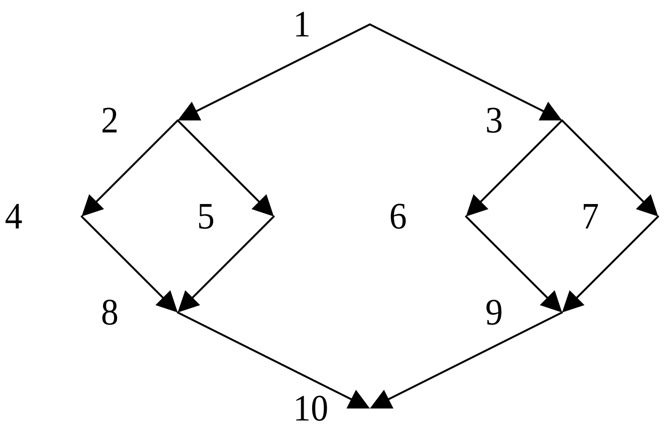
Computing Static Single Assignment Form

Computing SSA Form

1. Compute the Dominance Frontier (DF) from the Control Flow Graph (CFG).
2. Insert Φ-functions.
3. Rename variables.

Dominance Frontier (DF)

- Compute Dominator Relation, Dom, for each node, n, in CFG.

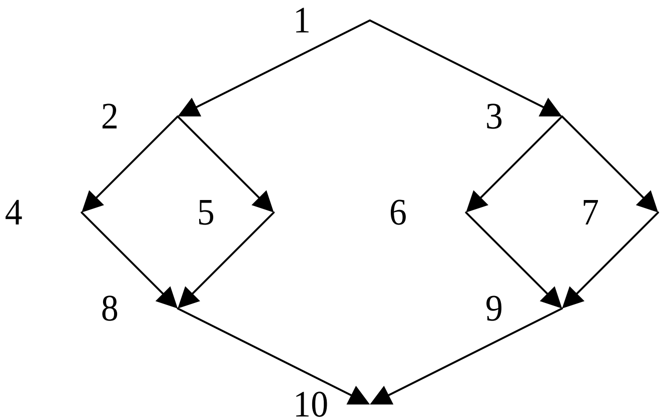


Dom(1) = {2,3,4,5,6,7,8,9,10}

Dom(3) = {6,7,9}

Dominance Frontier (DF)

- Compute Dominator Relation, Dom, for each node, n, in CFG.
- DF(n) contains the set of nodes that are *immediately reachable* in the CFG from Dom(n) but not in Dom(n).



Dom(1) = {2,3,4,5,6,7,8,9,10}

DF(1) = {}

Dom(3) = {6,7,9}

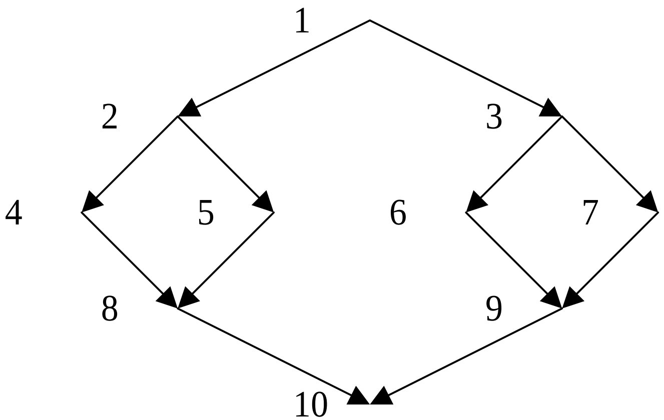
DF(3) = {10}

Dom(7) = {}

DF(7) ={9}

Dominance Frontier (DF)

- Compute Dominator Relation, Dom, for each node, n, in CFG.
- DF(n) contains the set of nodes that are *immediately reachable* in the CFG from Dom(n) but not in Dom(n).
- Iterated Dominance Frontier, IDF(n), is the transitive closure of DF(n).



Dom(1) = {2,3,4,5,6,7,8,9,10}

DF(1) = {}, IDF(1) = {}

Dom(3) = {6,7,9}

DF(3) = {10}, IDF(3) = {10}

Dom(7) = {}

DF(7) ={9}, IDF(7) = {9, 10}

Insert Φ-functions

- Φ-functions for v are inserted at the union of DF(v), for all of the assignments to v.

for each variable v in program:
  set phiLocs $\leftarrow$ {}
  for each definition d of variable v:
    phiLocs $\leftarrow$ phiLocs $\cup$ (IDF(BB(d)) $\cap$ Live(v))
  end
  insert Φ-functions for v at top of all phiLocs
end

Renaming Variables

- Each assignment to v is converted to an assignment to unique name v_i.
- Use depth first traversal of Dom.
- Keep stack of last seen name for v.
- Rename each uses with name on top of stack.

Incremental Static Single Assignment Form

Incremental SSA Form

- Delete an assignment to v - easy
- Add an assignment to v - moderately easy

Do not model move as delete and insert.

- Delete an edge - easy
- Add an edge - hard

Delete an Assignment to v_{del}

find the name v_{dom} of the variable v that is live before the assignment to v_{del}

for each use u of v_{del}

replace use u with v_{dom}

Add an Assignment to v_{new}

find the name v_{dom} of the variable v that is live before the assignment to v_{new}

let phiLoc $\leftarrow$ $(DF(v_{new}) - DF(v_{dom})) \cap Live(v_{dom})$

for each node n in phiLoc

if n contains a Φ-function that uses v_{dom}

then replace replace v_{dom} with v_{new}

else add Φ-function to n

run renaming algorithm starting at $BB(v_{dom})$ over $Dom(BB(v_{dom})) \cap Live(v_{dom})$

Move an Assignment

- A definition of variable can be moved to any location that dominates all of its uses.
- This is much faster than delete & insert.

Delete an Edge e

remove the edge e into basic block b

for each Φ-function in b

  delete the parameter that corresponds to e

end

Add an Edge e

- This is *hard.*
- In the NaturalBridge compiler we never add edges.
- We can grow single entry-multiple exit regions:
  - Procedure integration
  - Loop unrolling
- Use loop-closed SSA form.

Loop Closed SSA Form

- SSA form but with extra Φ-functions added at loop exits.
  - A special copy is added to each exit for each variable modified within the loop.
  - Loop exits become join nodes as the loop is replicated by unrolling.
  - The special copies are later turned into Φ-functions as exit edges are added into their blocks.
  - These extra Φ-functions gather these values together.

Loop Closed SSA Form

- The NaturalBridge version differs because we add special copy statements before SSA form is built.
  - Current unrolling code needs to be fixed since it gets out of ssa form and back in to build loop closed SSA form.
- We also add them in other places than loops:
  - Loop Exits — for loop unrolling
  - Exception Handlers — for procedure inlining
  - After conditionals — explained later

NaturalBridge INC

Tree-SSA-Dom

- This phase does three things:
  - Constant propagation
  - Value numbering
  - Branch forwarding

Problems With Tree-SSA-Dom

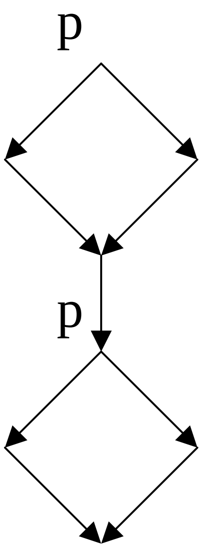
- Utility optimizations should be separate passes because they are run frequently.
  - You generally do not need the combined power or want to pay the cost.
  - Constant prop should be run frequently.
  - Branch forwarding should be run only twice.
- Constant propagation and value numbering needs to be a global iterative algorithms, not a single pass over the dominator tree.

Problems With Tree-SSA-Dom

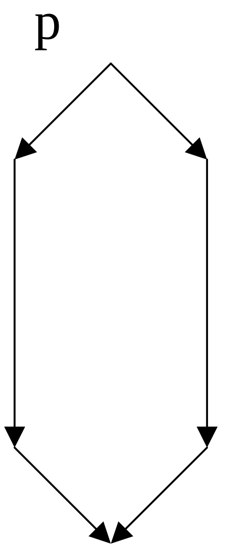
- Branch forwarding uses a poor algorithm.
  - This talk outlines a new algorithm.
- Constant Propagation in Tree-SSA-CCP needs some upgrading.
  - This talk outlines these problems.
- Value Numbering in Tree-SSA-Pre uses a poor algorithm.
  - This is about to be fixed by Danny Berlin.

Branch Forwarding

- Currently done in Tree-SSA-Dom.
  - gets out and back into SSA form.
- Should only be run twice at most.
  - before loop switching
  - near end of optimization



becomes



Getting Out then Back Into SSA

- Problems:
  - Expensive.
  - May not be correct.
  - End up with a lot of extra copy statements.
  - Loose all information attached to SSA variables.
    - Ranges, value numbers, aliasing information.
- Alternative:
  - Get in touch with me, zadeck@naturalbridge.com.

Branch Forwarding

1. Use value numbers from Tree-SSA-Pre to find fully redundant predicates.
2. Determine profitability.
3. Assess CFG structure.
4. Process Φ-functions and replicated code.
5. Cleanup.

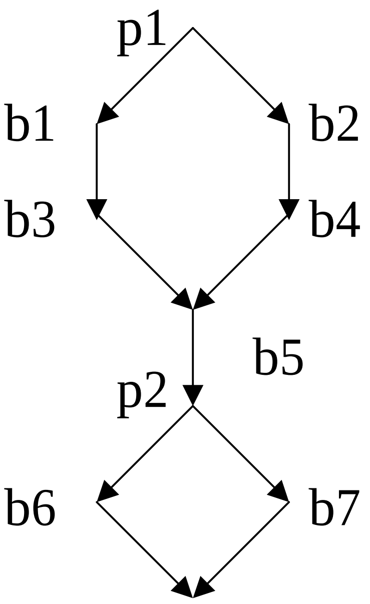
This will be done by Jeff Law.

Branch Forwarding
Find Fully Redundant Predicates

- Run Tree-SSA-Pre to produce value numbers for all expressions.
- Visit the statements in dominator tree order.
- Keep a dictionary of predicates seen so far.
- When the predicate, p2, at the bottom of the basic block, b, being visited matches a predicate, p1, in the dictionary, do steps in next 2 slides to see if this branch can be eliminated.

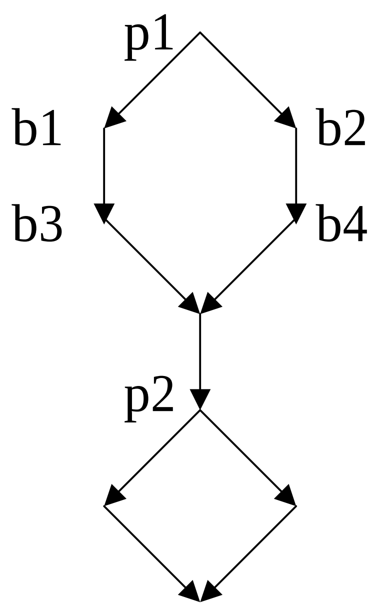
Branch Forwarding
Determine Profitability

- Walk backward from p2 counting the instructions and Φ-functions.
- If you reach a branch node, abort.
- Stop counting when you reach a join node.
- The forwarding operation will replicate all of the code from the join node to the predicate (b5).
  - If there is too much code here, do not do the forwarding.



Branch Forwarding
Assess CFG Structure

- p1 has two successors b1 and b2.
- The join node before p2 has predecessors of b3 and b4.
- It is safe to do the forwarding iff:
  - b1 dominates b3 and
  - b2 dominates b4
  - This domination test is safe no matter how complex the path is from b1 $\Rightarrow$ b3 or b2 $\Rightarrow$ b4.



Branch Forwarding

Process Φ-functions and Replicated Code

- Code in b5 will be copied to b6 and b7.
  - Use SSA add assignment operation.
  - Φ-functions turn into simple copy statements.
  - Regular code is replicated.

$i_3 \leftarrow \Phi(i_1, i_2)$
b5 $k_4 \leftarrow i_3 + 1$
if (p2)

b6 b7

becomes

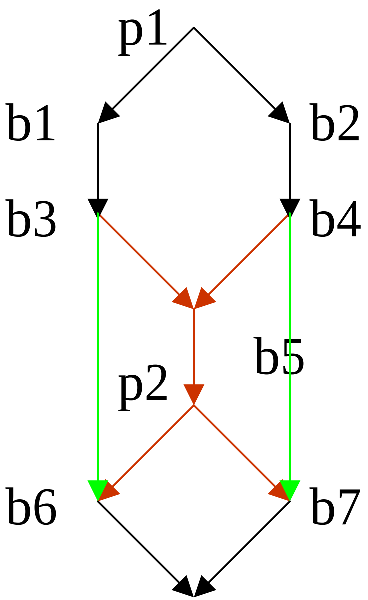
$i_3 \leftarrow \Phi(i_1, i_2)$
b5 $k_4 \leftarrow i_3 + 1$
if (p2)

$i_4 \leftarrow i_1$ $i_5 \leftarrow i_2$
b6 $k_5 \leftarrow i_4 + 1$ $k_6 \leftarrow i_5 + 1$ b7

Branch Forwarding

Cleanup

- Delete the code in b5.
  - There are no uses for any of the variables.
- Forward the edges around b5.
  - b3 -> b6
  - b4 -> b7
- Delete the edges associated with b5.



Problems with Tree-SSA-CCP

- Performance Issues:
  - Poor implementation of CFG in-edges
  - Locality control
  - Fast traversal of lattice
- Coverage Issues:
  - Richer lattice
  - Better information at conditionals

Tree-SSA-CCP Performance Issues
Poor Implementation of CFG In-Edges

- In-edges should be a vector not a linked list.
- Basic blocks for exception handlers may have hundreds of in-edges.
- Needs to mimic Φ-function in-edges.
- Savings:
  - space - no space saved for cfg itself but Φ-function does not need pointer to cfg edge
  - time - edges can be deleted in constant time
  - locality - vectors are compact
- To be done by Ben Elliston.

Tree-SSA-CCP Performance Issues

Locality Control

- The processing of the worklist should be ordered by depth-first number of dom tree.
- This will control the bouncing around in large functions.

- This may be done after measurement on large functions.

Tree-SSA-CCP Performance Issues
Fast Traversal of Lattice

- Most values are not constants:
  - especially true for subsequent executions
- In worst case, each operand is examined height of lattice –1 times.
- Execution should favor $\perp$ operations first.
  - this will drag dependant operations to $\perp$ skipping intermediate levels of the lattice.
- Already implemented by Danny Berlin.

Tree-SSA-CCP Coverage Issues
Richer Lattice

- Lattice models operation at Φ-function.
- CCP works as long as the lattice is bounded.
- Merges cannot raise value in lattice.
- Lattice will have 5 rather than 3 levels.
- Useful values to add:
  - constants: -2, -1, 0, 1, 2
  - halfRanges: [-2, $\top$], [$\top$, 5]
  - ranges: [-2..2], [5..10]
  - antiRanges: ~[-2..2], ~[-5..10]

Current CCP Lattice

- Top — $\top$
- Constants — SN … -3 -2 -1 0 1 2 3 … LN
- Bottom — $\perp$

Enhanced CCP Lattice

- Top

$\top$

- Constants

SN … -3 -2 -1 0 1 2 3 … LN

- Half Ranges

[SN..$\top$] [-1..$\top$] [0..$\top$][1..$\top$][LN.. $\top$]

[$\top$..SN] [$\top$..-1] [$\top$..0] [$\top$..1] [$\top$..LN]

- Ranges and AntiRanges

[SN..-1] [SN..0] [SN..1] [-1..0] [-1..1]
[-1..LN] [0..1] [0..LN][1..LN]

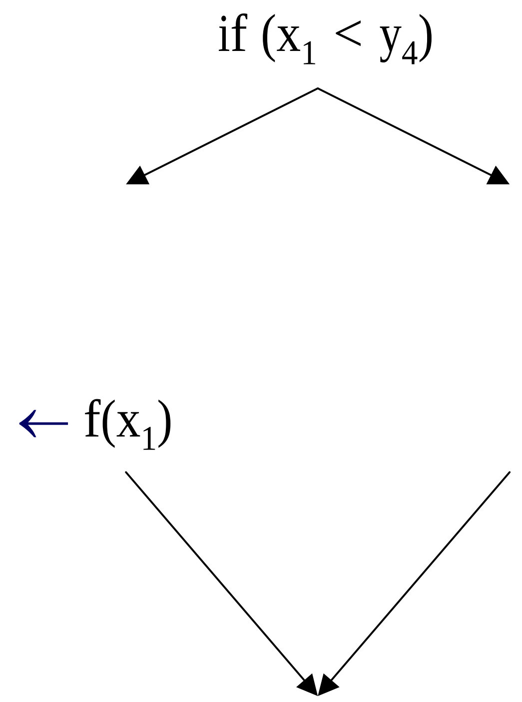
~[SN..-1] ~[SN..0] ~[SN..1] ~[-1..0] ~[-1..1]
~[-1..LN] ~[0..1] ~[0..LN]~[1..LN]

- Bottom

$\perp$

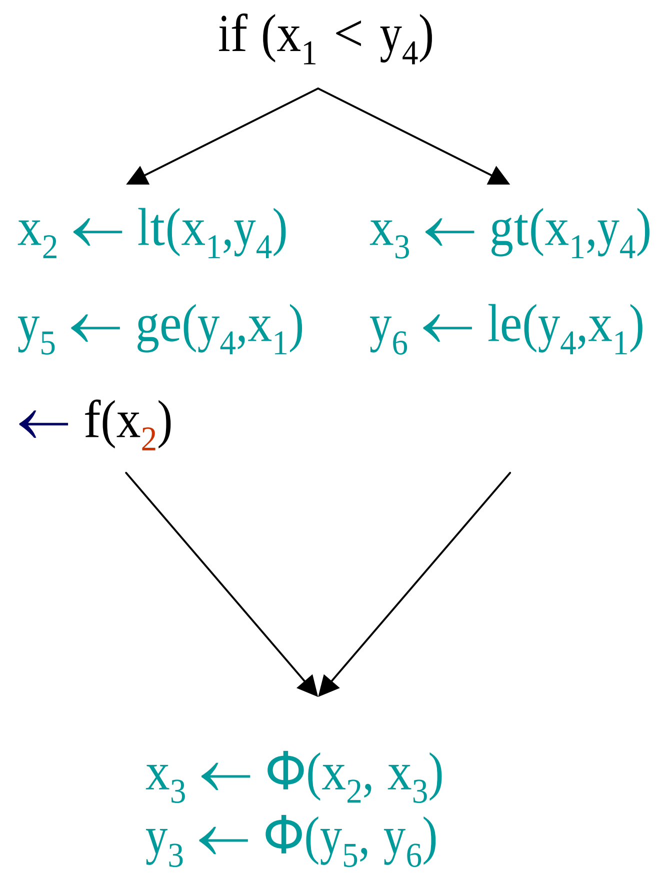
Tree-SSA-CCP Coverage Issues
Better Information at Conditionals

- Conditionals provide information about values.
- If y_4 is proven to be constant, we know something about x_1 on the true side of the branch.



Tree-SSA-CCP Coverage Issues
Better Information at Conditionals

- Conditionals provide information about values.
- If y_4 is proven to be constant, we know something about x_1 on the true side of the branch.
- Insert new live ranges for variables mentioned in test.
- New variables are placeholders for ranges and other information.
- Same trick as loop closed SSA.
- Info lost when getting out of SSA.



Tree-SSA-CCP Coverage Issues
Better Information at Conditionals

- Doing this allows redundant conditionals to be deleted.
  - null checks (~[0..0])
  - array bounds checks
  - mudflaps
  - user level checks
- This will be done by Diego Novilla

```
If (p != null)
  then {
     …
     if (p != null)
  }
```

Range Propagation

- Discovering range information is very different from constant propagation:

Constant Propagation:

- Optimistic
- Well defined fixed point
- Algorithmic

Range Analysis:

- Pessimistic
- Truncated Iteration
- Heuristic

Range Propagation

- Range analysis is currently done in Tree-SSA-Loop-nIter.
- This phase needs to be upgraded:
  - Should start with the output of Tree-SSA-CCP.
  - Ranges needed for other things than loop bounds.
  - Need to update the heuristics.
- This phase is generally one of the targets of the performance analysis crowd.
- Diego Novillo is planning to attack this soon.

Status

- Finished examination of all Tree SSA code.
- Finished examination of all Loop SSA code.
- Identified the small easy changes.

Next Steps

- Attack the aliasing implementation.
  - This is a place where the other developers have been sloppy.
  - Many passes ignore statements with non-trivial aliasing.
  - Aliasing is hard to understand.
- Upgrade the loop optimizations.
  - Some are translations from older RTL algorithms.
- Help anyone with SSA algorithm problems.