Advanced Compilers

CMPSCI 710

Spring 2003

Using SSA form

Emery Berger

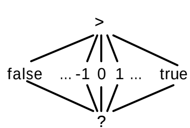
University of Massachusetts, Amherst

Topics

- Last time
  - Computing SSA form
- This time
  - Optimizations using SSA form
    - Constant propagation & dead code elimination
    - Loop invariant code motion

Constant Propagation

- Lattice for integer addition, multiplication, mod, etc.



Boolean Lattices, AND

AND	$\top$	*false*	*true*	$\bot$
$\top$	$\top$	*false*	$\top$	$\bot$
false	*false*	*false*	*false*	*false*
true	$\top$	*false*	*true*	$\bot$
$\bot$	$\bot$	*false*	$\bot$	$\bot$

- meet functions
  example: true AND **?**, false AND **>**

Boolean Lattices, OR

OR	$\top$	*false*	*true*	$\bot$
$\top$	$\top$	$\top$	*true*	$\bot$
false	$\top$	*false*	*true*	$\bot$
true	*true*	*true*	*true*	*true*
$\bot$	$\bot$	$\bot$	*true*	$\bot$

- meet functions
  example: true OR **?**, false OR **>**

Lattice for F -Nodes

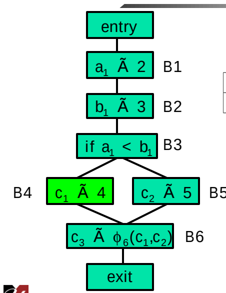
ϕ	$\top$	c_1	$\bot$
$\top$	$\top$	c_1	$\bot$
c_1	c_1	c_1	$\bot$
$c_2 \neq c_1$	c_2	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

- To propagate constants:
  if constant appears in conditional
  - Insert assignment on true branch

Constant Propagation Using SSA Form

- Initialize all expressions to >
- Two work lists:
  - CFG edges = (entry, x)
    - x = first reachable node
  - SSA edges = ∅
- Pick edge from either list until empty
  - Propagate constants when visiting either edge
  - When we visit a node:
    - ϕ-node: meet lattice values
    - others: add SSA successors, CFG successors

Sparse Conditional Constant Propagation Example



a_1	b_1	c_1	c_2	c_3

CFG edges	SSA edges

- Initialize all expressions to >
- Two work lists:
  - CFG edges = (entry, x)
  - SSA edges = ∅
- Pick edge from either list until empty
  - Propagate constants when visiting either edge
  - When we visit a node:
    - ϕ-node: meet lattice values
    - others: add SSA successors, CFG successors

Loop Optimizations

- Why optimize loops?
  - Loops = frequently-accessed code
    - regular patterns – can simplify optimizations
    - rule of thumb: loop bodies execute 10^{depth} times
    - optimizations pay off
- But why do we care if we aren't using FORTRAN?
  - Loops aren't just over arrays!
    - Pointer-based data structures
    - Text processing…

Loop Invariant Code Motion

- Classical loop optimization
  - avoids unnecessary computation

```
while (z < 1000)
  t = a + b
  z += t
```

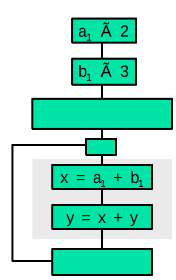
➔

```
t = a + b
while (z < 1000)
  z += t
```


Removing Loop Invariant Code

- Build SSA graph
- Simple test:
  - no operands in statement are defined by ϕ node or have definition inside loop
  - if match:
    - assign computation new temporary name and move to loop pre-header, and add assignment to temp
    - e.g., $l = r_1$ op r_2 becomes $t_1 = r_1$ op r_2; $l = t_1$

Loop Invariant Code Motion Example

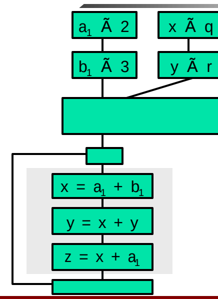


- Build SSA graph
- Simple test:
  - no operands in statement are defined by ϕ node or have definition inside loop
  - if match:
    - assign computation new temporary name and move to loop pre-header, and add assignment to temp
    - e.g., l = r1 op r2 becomes t1 = r1 op r2; l = t1

Finding More Invariants

- Build SSA graph
- If operands point to definition inside loop and definition is function of invariants (recursively)
  - replace as before

Loop Invariant Code Motion Example II



- Build SSA graph
- If operands point to definition inside loop and definition is function of invariants (recursively)
  - replace as before

Loop Induction Variables

- Loop induction variable:
  increases or decreases by constant amount inside loop
  - e.g., for (i = 0; i < 100; i++)
- Opportunity for:
  - *strength reduction*
    - e.g., j = 2 * i becomes j = j + 2
  - identifying *stride* of accesses for prefetching
    - e.g.: array accesses

Easy Detection of Loop Induction Variables

- Pattern match & check:
  - Search for "i = i + b" in loop
  - i is induction variable if no other assignment to i in loop
- Pros & Cons:
  - \+ Easy!
  - \- Does not catch all loop induction variables e.g., "i = a * c + 2"

Next Time

- Finding loop induction variables
- Strength reduction
- Read ACDI ch. 12, pp 333-342
- Project Design documents due
- March 13: project presentations
  - 5-10 minutes
  - 3 slides

Taxonomy of Induction Variables

- *basic* induction variable:
  - only definition in loop is assignment
    j := j § c, where c is loop invariant
- *mutual* induction variable:
  - definition is linear function of other induction variable i':
    - i = c1 * i' § c2
    - i = i' / c1 § c2
- *family* of basic induction variable j =
  set of induction variables i such that i always assigned linear function of j