Constant Propagation on SSA form

Advanced Compiler Techniques
2005
Erik Stenman
Virtutech

Constant Propagation

Safety
- Proves that name <u>always</u> has known value
- Specializes code around that value
  - Moves some computations to compile time (⇒ *code motion*)
  - Exposes some unreachable blocks (⇒ *dead code*)

Opportunity
- Value ≠ ⊥ signifies an opportunity

Profitability
- Compile-time evaluation is cheaper than run-time evaluation
- Branch removal may lead to block coalescing
  - If not, it still avoids the test & makes branch predictable

---

Sparse Constant Propagation Using SSA

∀ expression, e
  Value(e) ← { TOP if its value is unknown (or set by Φ-node); c_i if its value is known (the constant c_i); BOT if its value is known to vary }
WorkList ← Ø
∀ SSA edge s = <u,v>
  if Value(u) ≠ TOP then
    add s to WorkList

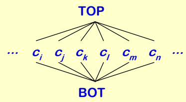
while (WorkList ≠ Ø)
  remove s = <u,v> from WorkList
  let o be the operation that uses v
  if Value(o) ≠ BOT then
    t ← result of evaluating o
    if t ≠ Value(o) then
      ∀ SSA edge <o,x>
        add <o,x> to WorkList

i.e., o is "a←b op v" or "a ←v op b"

Same result, fewer ∧ operations
Performs ∧ only at Φ nodes

Evaluating a Φ-node:
$\Phi(x_1,x_2,x_3, \ldots x_n)$ is
  $Value(x_1) \wedge Value(x_2) < \wedge Value(x_3)$
  $\wedge \ldots \wedge Value(x_n)$
Where
  TOP ∧ x = x   ∀ x
  $c_i \wedge c_j = c_i$ if $c_i = c_j$
  $c_i \wedge c_j$ = BOT   if $c_i \neq c_j$
  BOT ∧ x = BOT   ∀ x

Sparse Constant Propagation Using SSA



How long does this algorithm take to halt?
- Initialization is two passes
  - |ops| + 2 x |ops| edges
- Value(x) can take on 3 values
  - TOP, c_i, BOT
  - Each use can be on WorkList twice
  - 2 x |args| => 4 x |ops| evaluations, WorkList pushes & pops

This is an optimistic algorithm:
- Initialize all values to TOP, unless they are known constants
- Every value becomes BOT or c_i, unless its use is uninitialized

---

Sparse Constant Propagation
Optimism

```
i0 ← 12
while ( … )
   i1 ←  Φ(i0, i3)
   x ← i1 * 17
   j ← i1
   i2 ← …
     …
   i3 ← j
```

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)

Sparse Constant Propagation
Optimism

```
i0 ← 12
while ( … )
   i1 ←  Φ(i0, i3)
   x ← i1 * 17
   j ← i1
   i2 ← …
     …
   i3 ← j
```

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)

Sparse Constant Propagation

Optimism

```
i0 ← 12
while ( … )
  i1 ← Φ(i0, i3)
  x ← i1 * 17
  j ← i1
  i2 ← …
  …
  i3 ← j
```

Clear that *i* is always 12 at def of *x*

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)

Sparse Constant Propagation

Optimism

```
12   i0 ← 12
     while ( … )
⊥    i1 ← Φ(i0, i3)
⊥    x ← i1 * 17
⊥    j ← i1
⊥    i2 ← …
     …
⊥    i3 ← j
```

Pessimistic initializations

Leads to:

$i_1 \equiv 12 \wedge \perp \equiv \perp$
$x \equiv \perp * 17 \equiv \perp$
$j \equiv \perp$
$i_3 \equiv \perp$

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)

Sparse Constant Propagation

Optimism

```
12   i0 ← 12
     while ( … )
TOP  i1 ← Φ(i0, i3)
TOP  x ← i1 * 17
TOP  j ← i1
TOP  i2 ← …
     …
TOP  i3 ← j
```

Optimistic initializations

Leads to:

$i_1 \equiv 12 \wedge$ TOP $\equiv 12$
$x \equiv 12 * 17 \equiv 204$
$j \equiv 12$
$i_3 \equiv 12$
$i_1 \equiv 12 \wedge 12 \equiv 12$

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)

In general, optimism helps inside loops.

M.N. Wegman & F.K. Zadeck, Constant propagation with conditional branches, ACM TOPLAS, 13(2), April 1991, pages 181–210.

Sparse Conditional Constant Propagation

What happens when it propagates a value into a branch?

- TOP ⇒ we gain no knowledge.
- BOT ⇒ either path can execute.
- TRUE or FALSE ⇒ only one path can execute.

} But, the algorithm does not use this …

Working this into the algorithm.

- Use two worklists: SSAWorkList & CFGWorkList:
  - SSAWorkList determines values.
  - CFGWorkList governs reachability.
- Don't propagate into operation until its block is reachable.

Sparse Conditional Constant Propagation

```
SSAWorkList ← Ø
CFGWorkList ← n0
∀ block b
   clear b's mark
   ∀ expression e in b
      Value(e) ← TOP
```

Initialization Step

```
To evaluate a branch
   if arg is BOT then
      put both targets on CFGWorklist
   else if arg is TRUE then
      put TRUE target on CFGWorklist
   else if arg is FALSE then
      put FALSE target on CFGWorklist
To evaluate a jump
   place its target on CFGWorklist
```

```
while ((CFGWorkList ∪ SSAWorkList) ≠ Ø)
   while(CFGWorkList ≠ Ø)
      remove b from CFGWorkList
      mark b
      evaluate each Φ-function in b
      evaluate each op in b, in order
   while(SSAWorkList ≠ Ø)
      remove s = <u,v> from SSAWorkList
      let o be the operation that contains v
      t ← result of evaluating o
      if t ≠ Value(o) then
         Value(o) ← t
         ∀ SSA edge <o,x>
            if x is marked, then
               add <o,x> to SSAWorkList
```

Propagation Step

Sparse Conditional Constant Propagation

There are some subtle points:

- Branch conditions should not be TOP when evaluated.
  - Indicates an upwards-exposed use. (*no initial value - undefined*)
  - Hard to envision compiler producing such code.
- Initialize all operations to TOP.
  - Block processing will fill in the non-top initial values.
  - Unreachable paths contribute TOP to Φ-functions.
- Code shows CFG edges first, then SSA edges.
  - Can intermix them in arbitrary order. (*correctness*)
  - Taking CFG edges first may help with speed. (*minor effect*)

Sparse Conditional Constant Propagation

More subtle points:

- TOP * BOT → TOP
  - If TOP becomes 0, then 0 * BOT → 0.
  - This prevents non-monotonic behavior for the result value.
  - Uses of the result value might go irretrievably to 0.
  - Similar effects with any operation that has a "zero".
- Some values reveal simplifications, rather than constants
  - BOT * c_i → BOT, but might turn into shifts & adds (c_i = 2, BOT ≥ 0)
  - BOT**2 → BOT * BOT. (*vs. series or call to library*)
- `cbr TRUE → L1,L2` becomes `br → L1`
  - Method discovers this; it must rewrite the code, too!

Sparse Conditional Constant Propagation

Unreachable Code

```
i←17
if (i>0) then
   j1←10
else
   j2←20
j3←Φ(j1, j2)
k←j3*17
```

Optimism

- **Initialization to TOP is still important.**
- **Unreachable code keeps TOP.**
- **∧ with TOP has desired result.**

Sparse Conditional Constant Propagation

Unreachable Code

All paths execute

```
17   i←17
     if (i>0) then
10      j1←10
     else
20      j2←20
⊥    j3←Φ(j1, j2)
⊥    k←j3*17
```

Optimism

- **Initialization to TOP is still important.**
- **Unreachable code keeps TOP.**
- **∧ with TOP has desired result.**

Sparse Conditional Constant Propagation

Unreachable Code

With SCC marking blocks

```
17   i←17
     if (i>0) then
TOP     j1←10
     else
TOP     j2←20
TOP  j3←Φ(j1, j2)
170  k←j3*17
```

Optimism

- **Initialization to TOP is still important.**
- **Unreachable code keeps TOP.**
- **∧ with TOP has desired result.**

Sparse Conditional Constant Propagation

Unreachable Code

With SCC marking blocks

```
17   i←17
     if (i>0) then
10      j1←10
     else
TOP     j2←20   (struck out)
10   j3←Φ(j1, j2)
170  k←j3*17
```

Optimism

- **Initialization to TOP is still important.**
- **Unreachable code keeps TOP.**
- **∧ with TOP has desired result.**

Cannot get this any other way:

- **DEAD code cannot test (i > 0).**
- **DEAD marks j_2 as useful.**

Sparse Conditional Constant Propagation

Unreachable Code

With SCC marking blocks

```
17   i←17
     if (i>0) then
10      j1←10
     else
TOP     j2←20   (struck out)
10   j3←Φ(j1, j2)
170  k←j3*17
```

Optimism

- **Initialization to TOP is still important.**
- **Unreachable code keeps TOP.**
- **∧ with TOP has desired result.**

In general, combining two optimizations can lead to answers that cannot be produced by any combination of running them separately.
This algorithm is one example of that general principle.
Combining register allocation & instruction scheduling is another …

Using SSA Form for Optimizations

In general, using SSA conversion leads to:

- Cleaner formulations.
- Better results.
- Faster algorithms.

We've seen two SSA-based algorithms.

- Dead-code elimination.
- Sparse conditional constant propagation.