Combining Optimizations:
Sparse Conditional Constant Propagation
Reminders

1. You should all be working on your labs
2. Midterm due 5PM, Friday March 18 in DH 3080
   a) Covers lecture material up to but not including this lecture
   b) Two hour, closed-book, closed-notes exam
3. I will be out next week, beyond the range of email
   → I will let you know about substitute lecturers

Last class

• Finished SSA — out-of-SSA translation
• Looked at Sparse Simple Constant Propagation over SSA

This class

• Quick review of SSCP
• Sparse Conditional Constant Propagation over SSA
Using SSA — Sparse Constant Propagation

\[ \forall \text{ expression, } e \]
\[ \text{Value}(e) \leftarrow \begin{cases} \text{TOP} & \text{if its value is unknown} \\ \text{c}_i & \text{if its value is known} \\ \text{BOT} & \text{if its value is known to vary} \end{cases} \]
\[ \text{WorkList} \leftarrow \emptyset \]

\[ \forall \text{ SSA edge } s = <u,v> \]
\[ \text{if } \text{Value}(u) \neq \text{TOP} \text{ then} \]
\[ \text{add } s \text{ to WorkList} \]

while (WorkList \neq \emptyset)
\[ \text{remove } s = <u,v> \text{ from WorkList} \]
\[ \text{let } o \text{ be the operation that uses } v \]
\[ \text{if } \text{Value}(o) \neq \text{BOT} \text{ then} \]
\[ t \leftarrow \text{result of evaluating } o \]
\[ \text{if } t \neq \text{Value}(o) \text{ then} \]
\[ \text{Value}(o) \leftarrow t \]
\[ \forall \text{ SSA edge } <o,x> \]
\[ \text{add } <o,x> \text{ to WorkList} \]

\[ \text{i.e., } o \text{ is “} a \leftarrow b \text{ op } v \text{” or “} a \leftarrow v \text{ op } b \text{”} \]

\[ \text{Evaluating a } \emptyset \text{-node:} \]
\[ \emptyset(x_1, x_2, x_3, \ldots x_n) \text{ is} \]
\[ \text{Value}(x_1) \land \text{Value}(x_2) \land \text{Value}(x_3) \]
\[ \land \ldots \land \text{Value}(x_n) \]

Where
\[ \text{TOP} \land x = x \quad \forall \ x \]
\[ c_i \land c_j = c_i \quad \text{if } c_i = c_j \]
\[ c_i \land c_j = \text{BOT} \quad \text{if } c_i \neq c_j \]
\[ \text{BOT} \land x = \text{BOT} \quad \forall \ x \]

Same result, fewer \land operations

Performs \land only at \emptyset nodes
**Sparse Constant Propagation**

**Optimism**

Optimistic initializations

\[ i_0 \leftarrow 12 \]

\[ \text{while} \ (\ldots) \]

\[ i_1 \leftarrow \emptyset(i_0, i_3) \]

\[ x \leftarrow i_1 \times 17 \]

Leads to:

\[ i_1 = 12 \land \text{TOP} = 12 \]

\[ x = 12 \times 17 = 204 \]

\[ j = 12 \]

\[ i_3 = 12 \]

\[ i_1 = 12 \land 12 = 12 \]

Optimism

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

In general

- Optimism helps inside loops
- Largely a matter of initial value

Using SSA — Sparse Constant Propagation

How long does this algorithm take to halt?

- Initialization is two passes
  - \(|\text{ops}| + 2 \times |\text{ops}|\) edges
- Value(x) can take on 3 values
  - TOP, \(c_i\), BOT
  - Each use can be on WorkList twice
    - \(2 \times |\text{args}| = 4 \times |\text{ops}|\) evaluations, WorkList pushes & pops

Benefits of SSA

- Clean expression of the algorithm
- Actual improvements† in the algorithm’s cost

† Arguably, the improvements in cost are minor in this case.
Sparse Constant Propagation

What happens when SSCP 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 ...

Using this observation, we can create an algorithm that refines the notion of a feasible path in the CFG using knowledge of constants. That algorithm can go beyond the standard limits of data-flow analysis:

- Until a block can execute, treat it as unreachable
- Use optimistic initialization to allow the analysis to proceed with unevaluated blocks

The result is an analysis that can use limited symbolic evaluation to combine constant propagation with unreachable code elimination.
Sparse Conditional Constant Propagation

Can use constant-valued control predicates to refine the CFG

- If compiler knows the value of \( x \), it can eliminate either the then or the else case
  
  > (\( x > 0 \)) \( \Rightarrow \) \( B_2 \) is unreachable
  
  > (\( x > 0 \)) \( \Rightarrow \) \( y \) is 17 in \( B_3 \)

- This approach combines constant propagation with CFG reachability analysis to produce better results in each

- Example of Click’s notion of “combining optimizations”
  
  > This work both predated & motivated Click

Classic DFA assumes that all paths can be taken at runtime, including \((B_0, B_2, B_3)\)
**Sparse Constant Propagation**

To work this idea into the algorithm, make several modifications:

- Use two worklists:
  - SSAWorkList
    - Holds edges in the SSA graph
    - SSA worklist propagates changing values
  - CFGWorkList
    - Holds edges in the control-flow graph
    - CFG worklist propagates information on reachability

- Do not evaluate operations until block is reachable
- When it marks a block as reachable, must evaluate all operations
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 Ø-function in b
  evaluate each op o in b, in order
    ∀ SSA edge <o,x>
      if block(x) is marked
        add <o,x> to SSAWorklist
while(SSAWorkList ≠ Ø)
  remove s = <u,v> from WorkList
  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 block(x) is marked, then
        add <o,x> to SSAWorkList

Propagation Step

Note that the statement of this algorithm in EaC1e is badly mangled. Fixed in EaC2e.
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)
  > Hard to envision compiler producing such code

• Initialize Value for all expressions to TOP
  > Block processing will fill in the non-top initial values
  > Algorithm propagates values into marked blocks, but uses block processing to pull values into a block on 1\textsuperscript{st} evaluation
  > Unreachable paths contribute TOP to Ø-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 BOT
  - 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 (\( c_i = 2, \text{BOT} \geq 0 \))
  - Removes commutativity (reassociation)
  - BOT**2 → BOT * BOT (vs. series or call to library)

- \( \text{cbr TRUE} \rightarrow L_1, L_2 \) becomes \( \text{br} \rightarrow L_1 \)
  - Method discovers this; it must rewrite the code, too!
  - As stated, algorithm does not “eliminate” unreachable blocks
Sparse Conditional Constant Propagation

Unreachable Code

```
1   x₀ ← 1
   while (some condition)
   {
   ↓   x₂ ← Ø(x₀, x₅)
   if (x₂ > 0)
   17  x₃ ← 17
   else
   -273 x₄ ← -273
   ↓   x₅ ← Ø(x₃, x₄)
   ...
   }
   ...
   ...
```

Optimism

- Initialization to TOP is still important
- Unreachable code keeps TOP
- \( \wedge \) with TOP has desired result

Better example

All paths execute

Initial Values

-273

Final Values

1
Sparse Conditional Constant Propagation

Unreachable Code

1 \( x_0 \leftarrow 1 \)
\[ \text{while( some condition) } \]
\[ \{ \]
17 \( x_2 \leftarrow \emptyset(x_0, x_5) \)
\[ \text{if } (x_2 > 0) \]
17 \( x_3 \leftarrow 17 \)
else
17 \( x_4 \leftarrow 273 \)
17 \( x_5 \leftarrow \emptyset(x_3, x_4) \)
\[ \ldots \]
\[ \} \]
\[ \ldots \leftarrow x_5 \]

Corrected example

Optimism

- Initialization to TOP is still important
- Unreachable code keeps TOP
- \( \wedge \) 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

```
1  x_0 \leftarrow 1
   \text{while (some condition)}
   
   \{ 
   17  x_2 \leftarrow \emptyset(x_0, x_5) 
       \text{if (x_2 > 0)} 
       17  x_3 \leftarrow 17 
       \text{else} 
       273 x_4 \leftarrow 273 
   \}

17  x_5 \leftarrow \emptyset(x_3, x_4) 
   \ldots 
   \} 
   \ldots \leftarrow x_5 
```

Corrected example

```
With SCCP Marking Blocks
```

Final Values

```
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 allocation & scheduling is another ...
```

Optimism

- Initialization to TOP is still important
- Unreachable code keeps TOP
- $\land$ with TOP has desired result
The Role of SSA

What role does SSA play in SCCP?

- Can we formulate SCCP over def-use chains?
- Does SSA let the algorithm produce better results?
- Does SSA affect the running time of the algorithm?

The primary impact of SSA on this algorithm is make the statement of the algorithm cleaner and easier to understand.

- Minor reduction in number of meet operations
- Minor improvement in speed
- No difference in results