Cleaning up the CFG
Eliminating useless nodes & edges

COMP 512
Rice University
Houston, Texas
Spring 2009

The primary algorithm described in this lecture is due to Rob Shillner, when he worked on the MSCP at Rice. To our knowledge, the only published version of the algorithm is in Chapter 10 of EaC.

Dead Code Elimination

Three distinct problems

- Useless operations
  - Any operation whose value is not used in some visible way
  - Use the SSA-based mark/sweep algorithm (DEAD)

- Useless control flow
  - Branches to branches, empty blocks
  - Simple CFG-based algorithm (Shillner’s CLEAN)

- Unreachable blocks
  - No path from \( n_0 \) to \( b \) \( \Rightarrow \) \( b \) cannot execute
  - Simple graph reachability problem
Eliminating Useless Control Flow

The Problem
• After optimization, the CFG can contain empty blocks
• “Empty” blocks still end with either a branch or a jump
• Produces jump to jump, which wastes time & space
• Need to simplify the CFG & eliminate these

The Algorithm (CLEAN)
• Use four distinct transformations
• Apply them in a carefully selected order
• Iterate until done

Devised by Rob Shillner (1992), documented by John Lu (1994)
Eliminating Useless Control Flow

Transformations

Merging an empty block
- Empty \( B_1 \) ends in a jump
- Coalesce \( B_1 \) with \( B_2 \)
- Move \( B_1 \)’s incoming edges
- Eliminates extraneous jump
- Faster, smaller code

How does this happen?
- Eliminate operations in \( B_1 \)

How do we find it?
- Test for empty block

Eliminating empty blocks

Combining non-empty blocks

Coalescing blocks
- Neither block must be empty
- \( B_1 \) ends with a jump
- \( B_2 \) has 1 predecessor
- Combine the two blocks
- Eliminates a jump

How does this happen?
- Simplifying edges out of \( B_1 \)

How do we find it?
- Check target of jump \(|\text{preds}|\)

\( B_1 \) and \( B_2 \) should be a single basic block
If one executes, both execute, in linear order.
**Eliminating Useless Control Flow**

**Transformations**

- Jump to a branch
  - $B_1$ ends with jump, $B_2$ is empty
  - Eliminates pointless jump
  - Copy branch into end of $B_1$
  - Might make $B_2$ unreachable

**How does this happen?**

- Eliminating operations in $B_2$

**How do we find this?**

- Jump to empty block

**Hoisting branches from empty blocks**

**Putting the transformations together**

- Process the blocks in postorder
  - Clean up $B_i$’s successors before $B_i$
  - Simplifies implementation & understanding

- At each node, apply transformations in a fixed order
  - Eliminate redundant branch
  - Eliminate empty block
  - Merge block with successor
  - Hoist branch from empty successor

- May need to iterate
  - Postorder => unprocessed successors along back edges
  - Can bound iterations, but deriving tight bound is hard
  - Must recompute postorder between iterations
**Eliminating Useless Control Flow**

What about an empty loop?

- By itself, **CLEAN** cannot eliminate the loop
- Loop body branches to itself
  - Branch is *not* redundant
  - Doesn’t end with a jump
  - Hoisting does not help
- Key is to eliminate self-loop
  - Add a new transformation?
  - Then, \( B_1 \) merges with \( B_2 \)

New transformation must recognize that \( B_1 \) is empty. Presumably, it has code to test exit condition & (probably) increment an induction variable. This requires looking at code inside \( B_1 \), and doing some sophisticated pattern matching. This seems awfully complicated.
Eliminating Useless Control Flow

What about an empty loop?

- How to eliminate \(<B_1, B_1>\)?
  - Pattern matching?
  - Useless code elimination?
- What does DEAD do to \(B_1\)?
  - Remember, it is empty
  - Contains only the branch
  - \(B_1\) has only one exit
  - So, \(B_1 \notin RDF(B_2)\)
  - \(B_1\)'s branch is useless
  - DEAD rewrites it as a jump to \(B_2\)

Using SSA – Dead code elimination

Mark
for each op \(i\)
clear \(i\)'s mark
if \(i\) is critical then
mark \(i\)
add \(i\) to WorkList
while (Worklist \(\neq \varnothing\))
remove \(i\) from WorkList
(i has form “\(x \leftarrow y \text{ op } z\)”)
if def(\(y\)) is not marked then
mark def(\(y\))
add def(\(y\)) to WorkList
if def(\(z\)) is not marked then
mark def(\(z\))
add def(\(z\)) to WorkList
for each \(b \in RDF(block(i))\)
mark the block-ending branch in \(b\)
add it to WorkList

Sweep
for each op \(i\)
if \(i\) is not marked then
if \(i\) is a branch then
rewrite with a jump to \(i\)'s nearest useful post-dominator
if \(i\) is not a jump then
delete \(i\)

Notes:
- Eliminates some branches
- Reconnects dead branches to the remaining live code
- Find useful post-dominator by walking post-dom tree
- Entry & exit nodes are useful

Shillner added this clause
Eliminating Useless Control Flow

What about an empty loop?

- How to eliminate \(<B_1, B_2>\)?
  - Pattern matching?
  - Useless code elimination?
- What does DEAD do to \(B_1\)?
  - Remember, it is empty
  - Contains only the branch
  - \(B_1\) has only one exit
  - So, \(B_1 \notin RDF(B_2)\)
  - \(B_1\)'s branch is useless
  - DEAD rewrites it as a jump to \(B_2\)

DEAD converts it to a form where CLEAN handles it!

Eliminating Useless Control Flow

The Algorithm

```
CleanPass()
  for each block \(i\), in postorder
    if \(i\) ends in a branch then
      if both targets are identical then
        rewrite with a jump
    if \(i\) ends in a jump to \(j\) then
      if \(i\) is empty then
        merge \(i\) with \(j\)
      else if \(i\) has only one predecessor
        merge \(i\) with \(j\)
      else if \(j\) is empty & \(j\) has a branch then
        rewrite \(i\)'s jump with \(j\)'s branch
  Clean()
  until CFG stops changing
  compute postorder
  CleanPass()
```

Summary

- Simple, structural algorithm
- Limited transformation set
- Cooperates with DEAD
- In practice, quite fast
Eliminating Unreachable Code

The Problem
- Block with no entering edge
- Situation created by other optimizations

The Cure
- Compute reachability & delete unreachable code
- Simple mark/sweep algorithm on CFG
- Mark during computation of postorder, reverse postorder ...
- In MSCP, importing ILOC does this (every time)

Dead Code Elimination

Summary
- Useless Computations ⇒ DEAD
- Useless Control-flow ⇒ CLEAN
- Unreachable Blocks ⇒ Simple housekeeping

Other Techniques
- Constant propagation can eliminate branches
- Algebraic identities eliminate some operations
- Redundancy elimination
  > Creates useless operations, or
  > Eliminates them