Eliminating Useless Code & Control Flow

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapters 9 & 10 in EaC2e
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)
    → We will think of this in terms of **LIVE** information, rather than **SSA**
    → An analog of the Schorr-Waite marking algorithm, applied to code

• Useless control flow
  – Branches to branches, empty blocks
  – Simple **CFG**-based algorithm (CLEAN)

• Unreachable blocks
  – No path from $n_0$ to $b \Rightarrow b$ cannot execute
  – Simple graph reachability problem
Quick Vocabulary Lesson

In 412, we do not have time to teach the SSA construction

In this algorithm, we can approximate the complex analysis with 3 ideas

• **def**(x) takes us to the operation that defines x
  – Assume that we have a global algorithm to rename the code in a manner similar to the renaming done in local register allocation
  – The SSA construction (Chapter 9, EaC2e) does that renaming

• RDF(b), for a block b, is the set of blocks through which control must pass to reach b
  – Computation of RDF(b) is part of the dominance calculation
  – Easy to explain, hard to calculate

• Nearest useful postdominator for b is the nearest block that appears on every path from b to the procedure’s exit, and is marked as useful

All of this can be found in Chapters 9 & 10 of EaC2e

\(^1\) § 9.2.1, 9.3.2, 10.2.1 in EaC2e
Dead code elimination (The “DEAD” Algorithm)

Mark
for each op i
  clear i’s mark
  if i is critical then
    mark i
    add i to WorkList
while (Worklist ≠ ∅)
  remove i from WorkList
  (i has form “x←y 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 ∈ 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 postdominator
    if i is not a jump then
      delete i

Notes:
• Eliminates some branches
• Reconnects dead branches to the remaining live code
• Find useful postdominator by walking post-dominator tree
  > Entry & exit nodes are always “useful”
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 CLEAN Algorithm

- Use four distinct transformations
- Apply them in a carefully selected order
- Iterate until done

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

Transformations in CLEAN

Both sides of the branch target $B_i$
- Neither block must be empty
- Replace it with a jump to $B_i$
- Simple rewrite of last op in $B_1$

How does this happen?
- Rewriting other branches

How do we find it?
- Check each branch

Eliminating redundant branches

Branch, not a jump
Eliminating Useless Control Flow

Transformations in CLEAN

- 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

Merging an empty block

How does this happen?
- Eliminate operations in $B_1$

How do we find it?
- Test for empty block

Eliminating empty blocks
Eliminating Useless Control Flow

**Transformations in CLEAN**

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

**Combining non-empty blocks**

- $B_1$ and $B_2$ should be a single basic block
- If one executes, both execute, in linear order.

**How does this happen?**
- Simplifying edges out of $B_1$

How do we find it?
- Check target of jump $|\text{preds}|$
Eliminating Useless Control Flow

Transformations in CLEAN

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
Eliminating Useless Control Flow

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 $\Rightarrow$ unprocessed successors along back edges
  - Can bound iterations, but a deriving tight bound is hard
  - Must recompute postorder between iterations

Montonicity is not obvious
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?

\[
\begin{array}{c}
B_0 \\
\downarrow \\
B_1 \\
\downarrow \\
B_2
\end{array}
\]
Eliminating Useless Control Flow

What about an empty loop?

• How to eliminate \( \langle B_1, B_1 \rangle \) ?
  – 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 code elimination
(The “DEAD” Algorithm)

Mark
for each op i
  clear i’s mark
  if i is critical then
    mark i
    add i to WorkList
while (Worklist ≠ Ø)
  remove i from WorkList
  \((i \text{ has form } "x \leftarrow y \text{ op } z")\)
  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 \(b \in \text{ RDF(block(i))}\)
    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 always “useful”

Added to the algorithm by Shillingsburg
Eliminating Useless Control Flow

What about an empty loop?

• How to eliminate $\langle B_1, B_1 \rangle$?
  – 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, $B_1 \not\in RDF(B_2)$
  – $B_1$’s branch is useless
  – DEAD rewrites it as a jump to $B_2$

DEAD converts the empty loop 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 $j$ 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, its quite fast

How many calls to CleanPass are needed before CLEAN halts?
- Clearly a fixed point algorithm
- Answer is not obvious
Eliminating Useless Control Flow

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 a deriving tight bound is hard
  – Must recompute postorder between iterations

Montonicity is not obvious
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 did this (every time)
Dead Code Elimination

Summary

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

Other Transformations that eliminate dead code

• Constant propagation can eliminate some branch targets
• Algebraic identities & redundancy elimination make some operations useless or outright remove them (depends on implementation style)

Use of SSA Form

• DEAD used SSA form as a convenient way to get DEF-USE chains
• CLEAN operated on the CFG without much regard to contents of a block