Background

Code Motion algorithms come in three basic flavors

• Ad-hoc techniques, such as code hoisting & sinking, & GCSE
  > Opportunities found with data-flow information
  > Little attention paid to placement

• Data-flow techniques, such as partial redundancy elimination (PRE) & lazy code motion (LCM)
  > Opportunities found with data-flow information
  > Placement computed with scrupulous care

• Graphical techniques built around the structure of the CFG
  > Opportunities found using control dependence
  > Placement dictated by control dependence (structure of CFG)

This lecture looks at a technique from that third category

• Popularity of PRE/LCM has caused these techniques to be ignored
**Background**

Lazy Code Motion is (probably) the dominant technique

- Conditional control-flow inside loop causes problems
- LCM does not lengthen any path \(\text{(conservative not speculative)}\)

![Diagram of lazy code motion]

**Weaknesses of PRE/LCM**

- Control flow
- Moving assignments
- Speculative motion
- Iterated code motion

Cytron, Lowry, & Zadeck

- “Code motion of control structures in high-level languages”
- Paper predates the papers that define SSA form
- This paper includes most of the critical ideas of SSA form

We will recast it into modern terminology

**Loops**

- Algorithm operates on loops, inside out ...
  - Paper refers to intervals; think of them as SCCs of the CFG hanging from interval header \(\text{(loops, not maximal intervals)}\)
- Each interval (loop) has a landing pad on entry & on exit
  - Guarded region at the loop’s entry, another on exit path
- Traverse each interval in dominated topological order
  - Think of this as reverse postorder ...

Formally, a dominated topological order has the property that if \(n\) and \(m\) are both successors of \(l\) in topological order, and \(n \text{ dom} m\), then \(n\) precedes \(m\)
Landing Pads

A
B
C
D
E

Interval in the original CFG

Landing Pads

A
B
C
D
E

Interval in the original CFG
**Landing Pads**

Interval in the original CFG

Augmented with landing pads

- Landing pad for backward motion
- Landing pads for forward motion

**Code Motion**

Two algorithms

- **Strict** — guaranteed not to lengthen paths
  > Restrictions similar to PRE & LCM
- **Nonstrict** — can (and does) lengthen some paths
  > “Speculative” code motion

**Big Picture**

- Visit intervals in innermost to outermost order
- Copy movable control-flow code into entry or exit landing pad
- Copy movable statements into landing pad
- Interval is summarized and treated (in surrounding interval) as an atomic operation
- Need a carefully constructed name space (SSA names)
**Strict Algorithm**

∀ interval \( I \), in order from innermost to outermost
∀ statement \( s \) in \( I \), in reverse postorder

Test \( s \) to see if it is *movable*
If \( s \) is movable then
Copy \( s \) into the entry landing pad
If \( s \) is a control structure
then replace copy in loop with a bit test
else delete original statement from the loop

Need a usable test for *movability*

---

**Strict Algorithm**

\( S \) is *movable* if and only if
- \( s \) is not *control dependent* on any immovable test
- Definitions that reach uses in \( s \) originate outside the loop
  > i.e., \( s \) is loop invariant
- \( s \) defines \( x \) and the algorithm has seen neither a \( \phi \) for \( x \) in the loop nor an immovable definition for \( x \)

\( y \) is *control dependent* on \( x \) if and only if
- \( \exists \) a path from \( x \) to \( y \) such that every node in \( y \) are post-dominated by \( y \), and
- \( x \) is not post-dominated by \( y \)
Strict Algorithm

∀ interval \( I \), in order from innermost to outermost
∀ statement \( s \) in \( I \), in reverse postorder
  test each statement \( s \) to see if it is movable
  if \( s \) is movable & a control structure then
    copy \( s \) into the entry landing pad
    replace copy in loop with a bit test
  else if \( s \) is movable & not a control structure
    then move \( s \) to the equivalent place in entry landing pad
  clean up any unused control structures

Details
• Copy control-structure to the landing pad and redirect any loop
  exits to the bottom of the landing pad
• Unused control structures
  > No control dependent statements (DEAD + CLEAN)
  > Known conditional value (SCCP)

How can they just delete the assignments in the loop?
• LCM moves expressions but not assignments
  > It needs to ensure that the original name gets the invariant
    value on each iteration
  > Those assignments would make control structures useful
    … if it could move the assignment out of the control structure …

• CLZ moves assignments & deletes empty control structures
  > Remember their renaming scheme (SSA)?
  > It creates a unique name for the definition & makes this
    particular kind of motion safe!

One key rationale for SSA: the assignment cannot be killed,
so this kind of code motion becomes safe (1986 paper)
Non-strict Algorithm

Some control structures are not movable

- Their behavior varies from iteration to iteration
- Moving them might
  - Raise an exception that the unmovable guard might prevent
  - Lengthen the computation if the guard is always false

Nonetheless, moving them might be profitable ...

CLZ also give a nonstrict algorithm that moves these operations

- If \( s \) depends on an immovable test, it is placed (in landing pad) where it will execute independent of the outcome of that test
- Algorithm can still guard \( s \) with any movable tests ...

Removing Redundancy

Further improvement is possible

- After combining (hoisting), it needs no guard in landing pad
  - May render control-flow structure useless
- CLZ present new algorithm for “combining” such statements
  - Might just use hoisting, sinking, or adapt SSAPre
Reflections

- CLZ uses just one pass over each interval
  - Accounts for 2\textsuperscript{nd}, 3\textsuperscript{rd}, ... order effects \( (dto, rpo) \)
  - Would need to iterate LCM to achieve same results..
- SSA name space is critical to correctness
  - Allows motion of assignments
  - Allows nonstrict movement of invariants
- Other techniques can achieve similar effects to COMMON
  - Peel the first iteration of the loop & apply strong techniques for redundancy & for constant propagation \( (SCCP) \)
  - Hoisting, sinking, maybe a derivative of SSAPre
- SSA Pre \( (\text{Chow et al., PLDI 97}) \)
  - Produced same results as PRE or LCM on SSA
  - Maybe they should have tried for better results, a la CLZ
  - Peel each loop, use SSA Pre, then SCCP, DEAD, CLEAN

Significance

This paper is important

- Underpinnings and motivations of SSA, years before SSA paper
- Code motion of control structures is still not commonly done
  - Community has seized on LCM as right paradigm
  - CLZ algorithms challenge the wisdom of that opinion
  - The SSA name space is a critical part of the big picture
- This work deserves wider recognition and use
Background

Last lecture, we looked at Lazy Code Motion (LCM), which moves redundant and partially redundant code out of loops

- Conditional control-flow inside loop causes problems
- LCM does not lengthen any path (conservative not speculative)

Weaknesses of PRE/LCM

1. Control flow
2. Moving assignments
3. Speculative motion
4. Iterated code motion