Local vs. Global Optimizations

• Local: inside a single basic block
• Global: intra-procedurally, across basic blocks
• In all of these techniques, we often use the results of control-flow analysis and data-flow analysis

• Our objectives
  – Optimization: loop-invariant code motion
    • It really should be called “code motion for loop-invariant code”
  – Other examples of optimizations, no details
Local Common Subexpression Elimination

\[
\begin{align*}
a & = b + c \\
b & = a - d \\
c & = b + c \\
d & = a - d \\
\end{align*}
\]

\(a - d\) is a common subexpression: when evaluated the second time, it will produce the same value.

*Optimized code:*

\[
\begin{align*}
a & = b + c \\
b & = a - d \\
c & = b + c \\
d & = b \\
\end{align*}
\]

*if b is live*

\[
\begin{align*}
a & = b + c \\
d & = a - d \\
c & = d + c \\
\end{align*}
\]

*if b is not live*

*Question:* can we eliminate \(b + c\) with this approach?
Algebraic Identities

- **Arithmetic**
  
  \[
  x + 0 = 0 + x = x \\
  x - 0 = x \\
  x \times 1 = 1 \times x = x \\
  x / 1 = x
  \]

- **Strength reduction**: replace a more expensive operator with a cheaper one
  
  \[
  2 \times x = x + x \\
  x / 2 = 0.5 \times x
  \]

- **Constant folding**: evaluate expressions at compile time and use the result
  
  \[
  x = 2 \times 3.14 \text{ is replaced with } x = 6.28
  \]
  
  needed with symbolic constants (#define in C, final vars/fields in Java) or after constant propagation
Movement of Loop-Invariant Computations

• Motivation: avoid redundancy

a = … All instructions whose
b = … right-hand side operands have
c = … reaching definitions that are

\textit{start loop}

\textit{only} from outside the loop

\textit{end loop}

d = a + b

e = c + d

But, this also applies transitively.

Need an algorithm to compute
the set \textit{Invariant} of all
loop-invariant instructions
Complete Definition and Algorithm

• Add an instruction to **Invariant** if each operand on the right-hand side satisfies one of the following
  – It is a constant: e.g., -5 or 3.1415
  – All its reaching definitions are from outside of the loop

• Repeat until no further changes to **Invariant** are possible
  – Find an instruction that is not in **Invariant** but each operand on the right-hand side satisfies one of the following
    • It is a constant
    • All its reaching definitions from outside of the loop
    • It has exactly one reaching definition, and that definition is already in **Invariant**
  – Add this instruction to **Invariant**

• Use the use-def chains to find all reaching defs
The Last Condition

- A right-hand side operand has exactly one reaching def, and that def is already in *Invariant*
- Why not *two* reaching defs, both in *Invariant*?
  
  ```
  if (...) a = 5; else a = 6;
  b = a+1;
  ```

- Even though each definition of *a* is in *Invariant*, the value of *b* is not guaranteed to be the same for each iteration
Correctness Conditions

• Not every loop-invariant computation (i.e., member of \textit{Invariant}) can be moved

• Are the red instructions invariant? Can they be moved?

```
for (...) {
    a = 1; ...
    if (...) break;
    b = a+1; ...
}
```

\textit{changed to}

```
a = 1;
b = a+1;
for (...) {
    ... if (...) break; ...
}
```
Correctness Conditions

- Given: a loop-invariant instruction from $\text{Invariant}$
- Condition 1: the basic block that contains the instruction must dominate all exits of the loop — i.e., dominate nodes that are sources of loop-exit edges
- Remember dominance: $d \text{ dom } n$ means that very path from ENTRY to $n$ must go through $d$
- In the original program any exit from the loop will be preceded by the invariant instruction
  - i.e., it is safe to move this instruction out of the loop

```c
for (...) {
    a = 1; ...
    if (...) break;
    b = a+1; ...
}  \text{ both red instructions violate this condition}
```
Correctness Conditions

• Condition 2: to move instruction $a = \ldots$, this must be the only assignment to $a$ in the loop

```c
for (...) { a = 1; ... if (...) { ... a = 2; ... } ... b = a+1; ... }
```

```c
a = 1; for (...) { ... if (...) { ... a = 2; ... } ... b = a+1; ... }
```

Why is this transformation incorrect?
Correctness Conditions

• Condition 3: to move \( a = \ldots \), every use of \( a \) in the loop must be reached only by this definition of \( a \)
  – Condition 2 ensures that there does not exist another definition of \( a \) in the loop; Condition 3 guarantees that even if there are definitions of \( a \) outside of the loop, they do not reach any use of \( a \) in the loop

\[
a = 1; \ for \ (\ldots) \ \{ \ \ldots \ b = a+1; \ \ldots \ a = 2; \ \ldots \ \}
\]

\[
a = 1; \ a = 2; \ for \ (\ldots) \ \{ \ \ldots \ b = a+1; \ \ldots \ \}
\]

Why is this transformation incorrect?
Code Transformation

• First, create a preheader for the loop
  – Original CFG
    ![Original CFG Diagram]
  – Modified CFG
    ![Modified CFG Diagram]

• Next, consider all instructions in Invariant, in the order in which they were added to Invariant
  – Each instruction that satisfies the three conditions is added at the end of the preheader, and removed from its basic block
Potentially Necessary Pre-Transformation

• Recall that all exits of a loop should be dominated by the instruction we want to move (Condition 1)

• Consider \( \text{while}(y<0) \) \{ \( a = 1+2; \ y++; \) \}

L1: if \( y<0 \) goto L2;
    goto L3;
L2: \( a = 1+2; \)
    \( y = y + 1; \)
    goto L1;
L3: ...

\( a = 1+2 \) does not dominate the exit node B1

loop header is now B3 and \( a = 1+2 \) dominates the exit node B5
Other Global Optimizations [you are not responsible for this material]

- We have already have seen one simple form of code motion for loop-invariant computations
- Common subexpression elimination
- Copy propagation
- Dead code elimination
- Elimination of induction variables
  - Variables that essentially count the number of iterations around a loop
- Partial redundancy elimination
  - Powerful generalization of code motion and common subexpression elimination (Dragon book, Section 9.5)
Code fragment from quicksort:
i = m-1; j = n; v = a[n];
while(1) {
    do  i = i+1;  while  (a[i] < v);
    do  j = j–1;  while  (a[j] > v);
    if (i>=j) break;
    x=a[i]; a[i] = a[j]; a[j] = x;
}
x=a[i]; a[i] = a[n]; a[n] = x;
\[ i = m - 1 \]
\[ j = n \]
\[ t_1 = 4 \cdot n \]
\[ v = a[t_1] \]

B1

\[ i = i + 1 \]
\[ t_2 = 4 \cdot i \]
\[ t_3 = a[t_2] \]
\[ \text{if } (t_3 < v) \]

\[ t_4 = 4 \cdot j \]
\[ t_5 = a[t_4] \]
\[ \text{if } (t_5 > v) \]

B2

\[ j = j - 1 \]
\[ t_6 = 4 \cdot i \]
\[ x = a[t_6] \]
\[ t_7 = 4 \cdot i \]
\[ t_8 = 4 \cdot j \]
\[ t_9 = a[t_8] \]
\[ a[t_7] = t_9 \]
\[ t_{10} = 4 \cdot j \]
\[ a[t_{10}] = x \]
\[ \text{goto} \]

B3

\[ t_{11} = 4 \cdot i \]
\[ x = a[t_{11}] \]
\[ t_{12} = 4 \cdot i \]
\[ t_{13} = 4 \cdot n \]
\[ t_{14} = a[t_{13}] \]
\[ a[t_{12}] = t_{14} \]
\[ t_{15} = 4 \cdot n \]
\[ a[t_{15}] = x \]

B4

\[ \text{if } (i \geq j) \]

B5

Common subexpression elimination

Local redundancy in B5:
\[ t_7 = 4 \cdot i \text{ already available in } t_6 \]
\[ t_{10} = 4 \cdot j \text{ already available in } t_8 \]

Local redundancy in B6:
\[ t_{12} = 4 \cdot i \text{ already available in } t_{11} \]
\[ t_{15} = 4 \cdot n \text{ already available in } t_{13} \]

Followed by dead code elimination for:
\[ t_7, t_{10}, t_{12}, t_{15} \]
Common subexpression elimination

Global redundancy in B5:
- \( t_6 = 4 \cdot i \) already available in \( t_2 \)
- Can change \( x = a[t_6] \) and \( a[t_6] = t_9 \)
- \( t_8 = 4 \cdot j \) already available in \( t_4 \)
- Can change \( t_9 = a[t_8] \) and \( a[t_8] = x \)

Global redundancy in B6:
- \( t_{11} = 4 \cdot i \) already available in \( t_2 \)
- Can change \( x = a[t_{11}] \) and \( a[t_{11}] = x \)
- \( t_{13} = 4 \cdot n \) already available in \( t_1 \)
- Can change \( t_{14} = a[t_{13}] \) and \( a[t_{13}] = x \)
i = m - 1
j = n
t1 = 4 * n
v = a[t1]

i = i + 1
t2 = 4 * i
t3 = a[t2]
if (t3 < v)

j = j - 1
t4 = 4 * j
t5 = a[t4]
if (t5 > v)

Common subexpression elimination

Global redundancy in B5:
x = a[t2] already available in t3
Can change x = a[t2]
t9 = a[t4] already available in t5
Can change t9 = a[t4]

Global redundancy in B6:
x = a[t2] already available in t3
Can change x = a[t2]
t14 = a[t1] not available in v. Why?

B1
B2
true
B3
true

B4
true

B5

B6

x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto

x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
Copy propagation

Copy in B5: $x = t3$
Can replace $a[t4] = x$ with $a[t4] = t3$
Also $t9 = t5$: replace $a[t2] = t9$ with $a[t2] = t5$

Copy in B6: $x = t3$
Can replace $a[t1] = x$ with $a[t1] = t3$

Enables other optimizations
i = m - 1
j = n
t1 = 4 * n
v = a[t1]

i = i + 1
t2 = 4 * i
t3 = a[t2]
if (t3 < v)

j = j - 1
t4 = 4 * j
t5 = a[t4]
if (t5 > v)

if (i >= j)
  B4
  x = t3
t9 = t5
  a[t2] = t5
  a[t4] = t3
  goto

Dead code elimination
Variable x is **dead** immediately after B5 and B6
Need to use liveness analysis for this.
Assignments x = t3 in B5 and B6 are dead code
Note that we needed to do copy propagation first, to expose the “deadness” of x.

Same for t9 in B5
i = m - 1
j = n
t1 = 4 * n
v = a[t1]

B1

if (t3 < v)
B3

j = j - 1
t4 = 4 * j
t5 = a[t4]
if (t5 > v)
B4

i = i + 1
t2 = 4 * i
t3 = a[t2]

B2

true

if (i >= j)
B5

a[t2] = t5
a[t4] = t3
goto

B6

t14 = a[t1]
a[t2] = t14
a[t1] = t3

Induction variables and strength reduction

Induction variables in B2:
Each time i is assigned, its value increases by 1
Each time t2 is assigned, its value increases by 4
Can replace t2 = 4 * i with t2 = t2 + 4

Induction variables in B2:
Each time j is assigned, its value decreases by 1
Each time t4 is assigned, its value decreases by 4
Can replace t4 = 4 * j with t4 = t4 - 4
**Elimination of induction variables**

After initialization, \( i \) and \( j \) are used only in B4. Can replace \( i \geq j \) with \( t2 \geq t4 \) in B4. After this, \( i = i + 1 \) and \( j = j - 1 \) become dead code and can be eliminated.

In general, if there are two or more induction variables in the same loop, it may be possible to eliminate all but one of them.
Original program: for the worst-case input, \( \sim 18(n-m) \) instructions would be executed in the outer loop, with \( \sim 6(n-m) \) multiplications.

Optimized program: for the worst-case input, \( \sim 10(n-m) \) instructions would be executed in the outer loop, without any multiplications.