Compiler Optimizations

Chapter 8, Section 8.5
Chapter 9, Section 9.1.7
Local vs. Global Optimizations

• Local: inside a single basic block
  – Simple forms of common subexpression elimination, dead code elimination, strength reduction, etc.
  – May sometimes require the results of a dataflow analysis (e.g., Live Variables analysis)

• Global: intraprocedurally, across basic blocks
  – Code motion, more complex common subexpression elimination, etc.

• Interprocedural optimizations: across procedure boundaries
  – Tricky; not used often
  – Sometimes we do procedure inlining and use intraprocedural optimizations
Part 1: Local Optimizations

• Based on a DAG representation of a basic block
• DAG nodes are
  – The initial values of variables (before the basic block)
  – Nodes labeled with operations – e.g. +, −, etc.
• DAG construction
  – Traverse the sequence of instructions in the basic block
  – Maintain a mapping: variable $v \rightarrow$ DAG node that corresponds to the last write to this variable
  – Construct new nodes only when necessary

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

• In the previous example, we identified the common subexpression \( a - d \)
  — This can be used to optimize the code — eventually we will reassemble the basic block from this DAG

• Reverse topological sort traversal; if a node has multiple variables, write to one of them and then copy to the others; ignore vars that are not live

• Is \( b \) live at the end of the basic block?

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

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

\[
\begin{align*}
  if \ b \ is \ live \\
  if \ b \ is \ not \ live
\end{align*}
\]
Common Subexpression Elimination

• This approach does not always find all redundancies

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

There are no common subexpressions here, but in fact \(a\) and \(e\) both have the value \(b_0 + c_0\)
Dead Code Elimination

• Assumes the output of Live Variables analysis (aka liveness analysis)
• Consider the DAG together with the map of DAG node that have the last write of a variable
• If a DAG node has zero incoming edges, and has zero writes of live variables, it can be eliminated
  – And this may enable other removals
  – Example (from before): suppose that e and c are not live at the exit of the basic block
    • First we can remove the node for e
    • Then, the node for c
Algebraic Identities

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

• **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
  \[ x = 2 \times 3.14 \text{ is replaced with } x = 6.28 \text{ (e.g., due to symbolic constants with } \#\text{define in C, final in Java, etc.)} \]

• **Commutativity**
  – E.g. \( a = x \times y \) and \( b = y \times x \) have a common subexpression
Local vs. Global Optimizations

• Local: inside a single basic block
  – Simple forms of common subexpression elimination, dead code elimination, strength reduction, etc.
  – May sometimes require the results of a dataflow analysis (e.g., Live Variables analysis)

• Global: intraprocedurally, across basic blocks
  – **Code motion**, more complex common subexpression elimination, etc.

• Interprocedural optimizations: across procedure boundaries
  – Tricky; not used often
  – Sometimes we do procedure inlining and use intraprocedural optimizations
Part 2: Loop-Invariant Computations

• Motivation: again, avoid redundancy

\[ a = \ldots \]
\[ b = \ldots \]
\[ c = \ldots \]

```
start loop
```

\[ d = a + b \]
\[ e = c + d \]

```
end loop
```

All instructions whose right-hand side operands have reaching definitions that are only from outside the loop

But, this also applies transitively ...

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

• Add an instruction to $Inv$ 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

• After this initialization, repeat until no further changes to $Inv$ are possible
  – Find an instruction that is not in $Inv$ 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 $Inv$
  – Add this instruction to $Inv$
  – Note: use the ud-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 \textit{Inv}

• Why not \textit{two} reaching defs, both already in \textit{Inv}?
  
  \begin{verbatim}
  if (...) a = 5; else a = 6;
  b = a+1;
  \end{verbatim}

  Even though each definition of \texttt{a} is in \textit{Inv}, the value of \texttt{b} is not guaranteed to be the same for each iteration

• But: not every loop-invariant computation (i.e., member of \textit{Inv}) can be moved out of the loop
  
  \begin{verbatim}
  for (...) { a = 1; ... if (...) break; ... b = a+1; ... }
  \end{verbatim}

  \begin{verbatim}
  a = 1; b = a+1; for (...) { ... if (...) break; ... }
  \end{verbatim}
Conditions for the Actual Code Motion

• Given: a loop-invariant instruction from \( \text{Inv} \)

• Condition 1: the basic block that contains the instruction must dominate all exits of the loop
  – i.e., all nodes that are sources of loop-exit edges
  – This means that it is impossible to exit the loop before the instruction is executed
    • And thus, it is safe to move this instruction before the loop header

• Condition 2: to move instruction \( a = \ldots \), this must be the only assignment to \( a \) in the loop
  – \( \text{for (…)} \{ \ a = 1; \ \text{… if (…)} \{ \ \text{… a = 2; \ … } \} \ \text{… b = a+1; \ … } \} \)
  – \( a = 1; \ \text{for (…)} \{ \ \text{… if (…)} \{ \ \text{… a = 2; \ … } \} \ \text{… b = a+1; \ … } \} \)
Conditions for the Actual Code Motion

• Condition 3: to move instruction \( a = \ldots \), 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
    – \( a = 1; \text{for} (\ldots) \{ \ldots b = a+1; \ldots a = 2; \ldots \} \)
    – \( a = 1; a = 2; \text{for} (\ldots) \{ \ldots b = a+1; \ldots \} \)

• Given the subset of instructions from \( Inv \) that satisfy the three conditions from above, we can now modify the three-address code
Complications with Arrays

• Reaching definitions and ud-chains for arrays:
  – Any \( x[y] = z \) is treated as a “definition” of the entire array \( x \) (really, a possible def for each array element)
  – Any \( x = y[z] \) is treated as a “use” of the entire array \( y \) (really, a possible use for each array element)
  – This is very approximate ... (better approaches exist)

• Various issues
  \[
  a[x] = y \\
  a[z] = w \\
  u=a[v]
  \]
  – Both definitions of \( a \) are reaching for the use of \( a \)
    • Unless we can prove that \( z \) is always equal to \( x \)
Code Transformation

• First, create a preheader for the loop

– Original CFG

– Modified CFG

• Next, consider all instructions in \textit{Inv}, in the order in which they were added to \textit{Inv}
  – Each instruction that satisfies the three conditions is added at the end of the preheader, and removed from its basic block
Problem for While/For Loops

\[ S \rightarrow \textbf{while} \ (E) \ S_1 \]

\[ S.startLabel = \text{newLabel}() \]
\[ S.exitLabel = \text{newLabel}() \]
\[ S.code = \]
\[ S.startLabel \ |\ | \]
\[ E.code \ |\ | \]
\[ "if \ (! \ E.\\addr \ ) \ goto \ " \ S.exitLabel \ |\ | \]
\[ S_1.code \ |\ | \]
\[ "goto \ " \ S.startLabel \ |\ | \]
\[ S.exitLabel \]
Example

```c
while (i <= n) {
    x = 3*4;
    res *= x;
    i += 1;
}
...

_l1:
    _t1 = i <= n;
    if (!_t1) goto _l2;
    x = 3*4;
    _t2 = res * x; res = _t2;
    _t3 = i + 1; i = _t3;
    goto _l1;

_l2: ...
```

Condition 1: the basic block that contains the instruction must dominate all exits of the loop – violated!
The Fix

```c
__l1:
  _t1 = i <= n;
  if (!_t1) goto __l2;
  x = 3*4;
  _t2 = res * x; res = _t2;
  _t3 = i + 1; i = _t3;
  goto __l1;
__l2: ...
```

```c
__l1:
  _t1 = i <= n;
  if (!_t1) goto __l2;
  x = 3*4;
  _t2 = res * x; res = _t2;
  _t3 = i + 1; i = _t3;
  _t1 = i <= n;
  if (!_t1) goto __l2;
  goto __l1;
__l2: ...
```
Conceptually ...

while (i <= n) {
    x = 3*4;
    res *= x;
    i += 1;
}

...
Part 3: Other Global Optimizations

• We have already 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 (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 = \text{m} - 1 \\
j = n \\
t_1 = 4 \cdot n \\
v = a[t_1] \\
i = i + 1 \\
t_2 = 4 \cdot i \\
t_3 = a[t_2] \\
\text{if (} t_3 < v \text{)} \\
B1 \\

\text{true} \\

\begin{align*}
\text{B2} \\
i &= i + 1 \\
t_2 &= 4 \cdot i \\
t_3 &= a[t_2] \\
\text{if (} t_3 < v \text{)} \\
\end{align*} \\

j = j - 1 \\
t_4 = 4 \cdot j \\
t_5 = a[t_4] \\
\text{if (} t_5 > v \text{)} \\
B3 \\

\text{true} \\

\begin{align*}
\text{B4} \\
i &= i + 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} \\
\end{align*} \\

\begin{align*}
\text{B5} \\
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} \\
\end{align*} \\

\begin{align*}
\text{B6} \\
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 \\
\end{align*} \\

\text{Local redundancy in B5:} \\
t_7 = 4 \cdot i : \text{value already available in } t_6 \\
t_10 = 4 \cdot j : \text{value already available in } t_8 \\

\text{Common subexpression elimination} \\

22
\[
i = m - 1 \\
j = n \\
t_1 = 4 \times n \\
v = a[t_1]
\]

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

\[
j = j - 1 \\
t_4 = 4 \times j \\
t_5 = a[t_4] \\
\text{if } (t_5 > v)
\]

**Common subexpression elimination**

Optimization:
- \(t_7 = 4 \times i\) becomes \(t_7 = t_6\)
- \(t_{10} = 4 \times j\) becomes \(t_{10} = t_8\)
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)

B1

B2

B3

B4
if (i >= j)
true

B5

B6

Copy propagation (details later)
t7 = t6 is a copy instruction; can we replace t7 with t6?
t10 = t8 is a copy instruction; can we replace t10 with t8?
\begin{align*}
i &= m - 1 \\
j &= n \\
t1 &= 4n \\
v &= a[t1] \\
i &= i + 1 \\
t2 &= 4i \\
t3 &= a[t2] \\
\text{if (t3 < v)} \\
j &= j - 1 \\
t4 &= 4j \\
t5 &= a[t4] \\
\text{if (t5 > v)} \\
t6 &= 4i \\
x &= a[t6] \\
t7 &= t6 \\
t8 &= 4j \\
t9 &= a[t8] \\
a[t6] &= t9 \\
t10 &= t8 \\
a[t8] &= x \\
go \text{to} \\
t11 &= 4i \\
x &= a[t11] \\
t12 &= 4i \\
t13 &= 4n \\
t14 &= a[t13] \\
a[t12] &= t14 \\
t15 &= 4n \\
a[t15] &= x
\end{align*}

\text{Copy propagation (details later)}
\begin{align*}
a[t7] &= t9 \text{ becomes } a[t6] &= t9 \\
a[t10] &= x \text{ becomes } a[t8] &= t9
\end{align*}
Dead code elimination

**t7** is dead at the end of B5 (remember *Live Variables data-flow analysis?*) and is not used inside B5; so, **t7=t6** is dead code

**t10** is dead at the end of B5 and is not used inside B5; so, **t10=t8** is dead code
i=m-1
j=n
t1= 4*n
v=a[t1]

B1

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

B2

true

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

B3

true

B4

if (i>=j)

B5

t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
goto

B6

t11=4*i
x=a[t11]
t12=4*i
t13=4*n
t14=a[t13]
a[t12]=t14
t15=4*n
a[t15]=x
\[ i = m - 1 \]
\[ j = n \]
\[ t_1 = 4 \cdot n \]
\[ v = a[t_1] \]

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

\[ j = j - 1 \]
\[ t_4 = 4 \cdot j \]
\[ t_5 = a[t_4] \]
\[ \text{if } (t_5 > v) \]

\[ \text{true} \]

\[ \text{Common subexpression elimination} \]
Local redundancy in B6:
\[ t_{12} = 4 \cdot i : \text{value already available in } t_{11} \]
\[ t_{15} = 4 \cdot n : \text{value already available in } t_{13} \]
Same optimizations as was done for B5
\[ i = m - 1 \]
\[ j = n \]
\[ t_1 = 4 \times n \]
\[ v = a[t_1] \]

**B1**

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

**B2**

\[ j = j - 1 \]
\[ t_4 = 4 \times j \]
\[ t_5 = a[t_4] \]
\[ \text{if } (t_5 > v) \]

**B3**

\[ \text{true} \]

**B4**

\[ \text{true} \]

\[ t_6 = 4 \times i \]
\[ x = a[t_6] \]
\[ t_8 = 4 \times j \]
\[ t_9 = a[t_8] \]
\[ a[t_6] = t_9 \]
\[ a[t_8] = x \]

**B5**

\[ \text{true} \]

**B6**

\[ t_{11} = 4 \times i \]
\[ x = a[t_{11}] \]
\[ t_{13} = 4 \times n \]
\[ t_{14} = a[t_{13}] \]
\[ a[t_{11}] = t_{14} \]
\[ a[t_{13}] = x \]
\text{Common subexpression elimination}

Global redundancy in B5:
- \( t6=4*i \) : value already available in \( t2 \)
- \( t8=4*j \) : value already available in \( t4 \)
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,
copy propagation, dead code elimination:

x=a[t6] becomes x=a[t2]
a[t6]=t9 becomes a[t2]=t9
t9=a[t8] becomes t9=a[t4]
a[t8]=x becomes a[t4]=x
t6 and t8 are eliminated
i=m-1
j=n
t1= 4*n
v=a[t1]

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

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

if (i>=j) true

x=a[t2]
t9=a[t4]
a[t2]=t9
t11=4*i
x=a[t11]
t13=4*n
x=a[t13]
t14=a[t13]
t11=t14
goto

a[t11]=t14
a[t13]=x
i=m-1
j=n
t1=4*n
v=a[t1]

if (t3<v)
   B2
   true
   B1

j=j-1
t4=4*j
t5=a[t4]
if (t5>v)
   B3
   true
   B2

if (i>=j)
   B4
   true
   B3

x=a[t11]
t9=a[t4]
a[t2]=t9
a[t4]=x
goto

B5

Common subexpression elimination, copy propagation, dead code elimination:

x=a[t11] becomes x=a[t2]
a[t11]=t14 becomes a[t2]=t14
t14=a[t13] becomes t14=a[t1]
a[t13]=x becomes a[t1]=x
t11 and t13 are eliminated
\[i = m - 1 \]
\[j = n \]
\[t_1 = 4 \times n \]
\[v = a[t_1] \]

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

\[j = j - 1 \]
\[t_4 = 4 \times j \]
\[t_5 = a[t_4] \]
\[\text{if (} t_5 > v \text{)} \]

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

\[x = a[t_2] \]
\[t_9 = a[t_4] \]
\[a[t_2] = t_9 \]
\[a[t_4] = x \]
\[\text{goto} \]

\[x = a[t_2] \]
\[t_{14} = a[t_1] \]
\[a[t_2] = t_{14} \]
\[a[t_1] = x \]
Additional common subexpressions?

x=a[t2]: value already available in t3

a[t4]=x becomes a[t4]=t3
a[t1]=x becomes a[t1]=t3
x is eliminated
i = m - 1
j = n
t1 = 4 * n
v = a[t1]

i = i + 1
t2 = 4 * i
t3 = a[t2]
if (t3 < v)
B2

j = j - 1
t4 = 4 * j
t5 = a[t4]
if (t5 > v)
B3

if (i >= j)
B4

B5

t9 = a[t4]
a[t2] = t9
a[t4] = t3
goto

B6

t14 = a[t1]
a[t2] = t14
a[t1] = t3
\(i = m-1\)
\(j = n\)
\(t1 = 4 \times n\)
\(v = a[t1]\)

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

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

\(t9 = a[t4]\)
\(a[t2] = t9\)
\(a[t4] = t3\)
goto

Additional common subexpressions?

\(t9 = a[t4]\) : value already available in \(t5\)

\(a[t2] = t9\) becomes \(a[t2] = t5\)
\(t9\) is eliminated
\[i = m - 1\]
\[j = n\]
\[t_1 = 4 \times n\]
\[v = a[t_1]\]

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

\[j = j - 1\]
\[t_4 = 4 \times j\]
\[t_5 = a[t_4]\]
\[\text{if } (t_5 > v)\]

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

\[a[t_2] = t_5\]
\[a[t_4] = t_3\]
\[\text{goto} t_{14} = a[t_1]\]

\[t_{14} = a[t_1]\]
\[a[t_2] = t_{14}\]
\[a[t_1] = t_3\]
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)

a[t2]=t5
a[t4]=t3
goto t14=a[t1]
a[t2]=t14
a[t1]=t3

B1

true

B2

true

B3

true

B4

if (i>=j)

B5

true

B6

Additional common subexpressions?

\texttt{t14=a[t1]}: \text{value already available in } v?\notag

\text{Not really ...}
\begin{align*}
i &= m - 1 \\
j &= n \\
t_1 &= 4 \times n \\
v &= a[t_1] 
\end{align*}

\begin{align*}
i &= i + 1 \\
t_2 &= 4 \times i \\
t_3 &= a[t_2] \\
\text{if} \ (t_3 < v) & \quad \text{true} 
\end{align*}

\begin{align*}
j &= j - 1 \\
t_4 &= 4 \times j \\
t_5 &= a[t_4] \\
\text{if} \ (t_5 > v) & \quad \text{true} 
\end{align*}

\text{Induction variables}

\text{Induction variables in B2:}
Each time \( i \) is assigned, its value increases by 1
Each time \( t_2 \) is assigned, its value increases by 4

\text{Induction variables in B3:}
Each time \( j \) is assigned, its value decreases by 1
Each time \( t_4 \) is assigned, its value decreases by 4
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)
a[t2] = t5
a[t4] = t3
goto

B1
B2
true
B3
true
B4
true
B5
B6

Strength reduction: use +/- instead of *
Induction variables in B2:
Can replace t2 = 4 * i with t2 = t2 + 4

Induction variables in B3:
Can replace t4 = 4 * j with t4 = t4 - 4
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.
(start detour) Another Dataflow Analysis

- **Copy propagation**: for \( x = y \), replace subsequent uses of \( x \) with \( y \), as long as \( x \) and \( y \) have not changed along the way
  - Creates opportunities for **dead code elimination**: e.g., after copy propagation we may find that \( x \) is not live

```
b=a
c=4*b
if (c>b)
    true
    d=b+2
    B2
B3
e=a+b
    EXIT
ENTRY
```

```
b=a
c=4*a
if (c>a)
    true
    d=a+2
    B2
B3
e=a+a
    EXIT
ENTRY
```

1) Dead code elimination: \( b=a \)
2) Strength reduction: \( e=a+a \) use left shift instead of addition
Formulation as a System of Equations

• For each CFG node $n$ (assume nodes = instructions)

\[
\text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} \text{OUT}[m] \\
\text{OUT}[\text{ENTRY}] = \emptyset
\]

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

– $\text{IN}[n]$ is a set of copy instructions $x=y$ such that neither $x$ nor $y$ is assigned along any path from $x=y$ to $n$

– $\text{GEN}[n]$ is
  • A singleton set containing the copy instruction, when $n$ is indeed a copy instruction
  • The empty set, otherwise

– $\text{KILL}[n]$: if $n$ assigns to $x$, kill every $y=x$ and $x=y$

– Note that we must use intersection of $\text{OUT}[m]$
Worklist Algorithm (end detour)

\[ \text{IN}[n] = \text{the set of all copy instructions}, \text{ for all } n \]

Put the successor of ENTRY on worklist

While (worklist is not empty)

1. Remove a CFG node \( m \) from the worklist
2. \( \text{OUT}[m] = (\text{IN}[m] - \text{KILL}[m]) \cup \text{GEN}[m] \)
3. For each successor \( n \) of \( m \)
   
   \[ \text{old} = \text{IN}[n] \]
   
   \[ \text{IN}[n] = \text{IN}[n] \cap \text{OUT}[m] \]
   
   If \( \text{old} \neq \text{IN}[n] \) add \( n \) to worklist

In Reaching Definitions, we initialized \( \text{IN}[n] \) to the empty set; here we cannot do this, because of \( \text{IN}[n] = \text{IN}[n] \cap \text{OUT}[m] \)

• Here the “merge” operation is \textit{set intersection}
• In Reaching Definitions, “merge” is \textit{set union}
Part 4: Loop Optimizations

- Loops are important for performance
- Parallelization
  - May need scalar expansion
- Loop peeling
- Loop unrolling
- Loop fusion
- Many more [CSE 5441 - Introduction to Parallel Computing]
  - Loop permutation (interchange)
  - Loop distribution (fission)
  - Loop tiling
  - Loop skewing
  - Index set splitting (generalization of peeling)
Loop Parallelization

• When all iterations of a loop are independent of each other

```c
for ( i = 0 ; i < 4096 ; i++ )
c[i] = a[i] + b[i];
```

• Needs some form of loop dependence analysis, which often involves reasoning about arrays

• May require enabling pre-transformations to make it parallel (e.g., scalar expansion or privatization)

```c
for ( i = 0 ; i < 4096 ; i++ ){
t = a[i] + b[i];
c[i] = t*t; }
```

Scalar expansion example

```c
double tx[4096];
for ( i = 0 ; i < 4096 ; i++ ){
tx[i] = a[i] + b[i];
c[i] = tx[i]*tx[i]; }
t = tx[4095];
```
Loop Peeling

- Goal: extract the first (or last) iteration
  - E.g. wraparound variables for cylindrical coordinates

```
j = N;
for (i = 0; i < N; i++) {
    b[i] = (a[i] + a[j]) / 2;
    j = i;}
```

- Peel-off the first iteration, then do induction variable analysis and copy propagation

```
j = N;
if (N>=1) {
b[0] = (a[0] + a[j]) / 2;
j = 0;
for (i = 1; i < N; i++) {
b[i] = (a[i] + a[j]) / 2;
j = i;}
}
```

```
if (N>=1) {
b[0] = (a[0] + a[N]) / 2;
for (i = 1; i < N; i++) {
b[i] = (a[i] + a[i-1]) / 2;
}
// now we can do unrolling
```
Loop Unrolling

• Loop unrolling: extend the body

```bash
for ( i = 0 ; i < 4096 ; i += 2 ) {
  c[i] = a[i] + b[i];
  c[i+1] = a[i+1] + b[i+1];
}
```

- Reduces the “control overhead” of the loop: makes the loop exit test \( i < 4096 \) execute less frequently
- Hardware advantages: instruction-level parallelism; fewer pipeline stalls
- Issue: loop bound may not be a multiple of unroll factor
- Problem: high unroll factors may degrade performance due to register pressure and spills (more later)
Loop Fusion

• Merge two loops with compatible bounds
  
  for ( i = 0 ; i < N ; i++ )
  c[i] = a[i] + b[i];
for ( j = 0 ; j < N ; j++)
  d[j] = c[j] * 2;

  for ( k = 0 ; k < N ; k++ ) {
    c[k] = a[k] + b[k];
    d[k] = c[k] * 2;
  }

  — Reduces the loop control overhead (i.e., loop exit test)
  — May improve cache use (e.g. the reuse of \(c[k]\) above) – especially producer/consumer loops \(\downarrow\) capacity misses
  — Fewer parallelizable loops and increased work per loop: reduces parallelization overhead (cost of spawn/join)
  — If the loop bounds are “±1 off” – use peeling
  — Not always legal – need loop dependence analysis