Code Optimization

Code Optimization and Phases

- Front End
- Code Optimizer
- Code Generator
- Control-Flow Analysis
- Data-Flow Analysis
- Transformations
Optimization

- Code produced by standard algorithms can often be made to run faster, take less space or both
- These improvements are achieved through transformations called optimizations
- Compilers that apply these transformations are called optimizing compilers
- It is especially important to optimize frequently executed parts of a program
  - The 90/10 rule
  - Profiling can be helpful, but compilers do not typically have sample input data
  - Inner loops tend to be good candidates for optimization

Criteria for Transformations

- A transformation must preserve the meaning of a program
  - Cannot change the output produced for any input
  - Can not introduce an error
- Transformations should, on average, speed up programs
- Transformations should be worth the effort
Beyond Optimizing Compilers

- Improvements can be made at various phases
- Source code:
  - Algorithmic transformations can produce spectacular improvements
  - Profiling can be helpful to focus a programmer's attention on important code
- Intermediate code:
  - Compiler can improve loops, procedure calls, and address calculations
  - Typically only optimizing compilers include this phase
- Target code:
  - Compilers can use registers efficiently
  - Peephole transformation can be applied

Peephole Optimizations

- A simple technique for locally improving target code (can also be applied to intermediate code)
- The peephole is a small, moving window on the target program
- Each improvement replaces the instructions of the peephole with a shorter or faster sequence
- Each improvement may create opportunities for additional improvements
- Repeated passes may be necessary
Redundant-Instruction Elimination

- Redundant loads and stores:
  1. lw $t0, a
  2. sw $t0, a

- Unreachable code:
  #define debug 0
  if (debug) {
    /* print debugging information */
  }

Flow-of-Control Optimizations

- Jumps to jumps, jumps to conditional jumps, and conditional jumps to jumps are not necessary

- Jumps to jumps example:
  - The following replacement is valid:
    
    \[
    \begin{array}{c}
    \text{goto L1} \\
    \text{...} \\
    \text{L1: goto L2}
    \end{array}
    \hspace{1cm}
    \begin{array}{c}
    \text{goto L2} \\
    \text{...} \\
    \text{L1: goto L2}
    \end{array}
    \]

  - If there are no other jumps to L1 and L1 is preceded by an unconditional jump, the statement at L1 can be eliminated

- Jumps to conditional jumps and conditional jumps to jumps lead to similar transformations
Other Peephole Optimizations

- A few algebraic identities that occur frequently (such as \( x := x + 0 \) or \( x := x \times 1 \)) can be eliminated.
- Reduction in strength replaces expensive operations with cheaper ones:
  - Calculating \( x \times x \) is likely much cheaper than \( x^2 \) using an exponentiation routine.
  - It may be cheaper to implement \( x \times 5 \) as \( x \times 4 + x \).
- Some machines may have hardware instructions to implement certain specific operations efficiently:
  - For example, auto-increment may be cheaper than a straightforward \( x := x + 1 \).
  - Auto-increment and auto-decrement are also useful when pushing into or popping off of a stack.

Optimizing Intermediate Code

- This phase is generally only included in optimizing compilers.
- Offers the following advantages:
  - Operations needed to implement high-level constructs are made explicit (i.e., address calculations).
  - Intermediate code is independent of target machine; code generator can be replaced for different machine.
- We are assuming intermediate code uses three-address instructions.
**Quicksort in C**

```c
void quicksort(int m, int n)
{ int i, j, v, x;

    if (n <= m) return;
    /* Start of partition code */
    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;
    /* End of partition code */
    quicksort(m, j); quicksort(i+1, n);
} /* quicksort */
```

**Three-Address Code**

- Find the basic blocks
- Draw the control flow graph

```
(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]
(5) i := i+1
(6) t2 := 4*i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```
Control Flow Graph

- Nodes represent basic blocks
- The initial node is the block whose leader is the first statement
- There exists an edge from Bi to Bj if:
  - There is a conditional or unconditional jump from the last statement in Bi to the first statement in Bj
  - Bj immediately follows Bi in the order of the program, and Bi does not end in an unconditional jump

Flow Graph
Local vs. Global Transformations

- Local transformations involve statements within a single basic block
- All other transformations are called global transformations
- Local transformations are generally performed first
- Many types of transformations can be performed either locally or globally

Common Subexpressions

- E is a common subexpression if:
  - E was previously computed
  - Variables in E have not changed since previous computation
- Can avoid recomputing E if previously computed value is still available
- Dags are useful to detect common subexpressions
Local Common Subexpressions

\[
\begin{align*}
t6 &:= 4*i \\
x &:= a[t6] \\
t7 &:= 4*i \\
t8 &:= 4*j \\
t9 &:= a[t8] \\
a[t7] &:= t9 \\
t10 &:= 4*j \\
a[t10] &:= x \\
goto B2
\end{align*}
\]

Global Common Subexpressions

\[
\begin{align*}
i &:= m-1 \\
j &:= n \\
t_1 &:= 4*n \\
v &:= a[t_1] \\
1 &:= i+1 \\
t_2 &:= 4*i \\
t_3 &:= a[t_2] \\
if \ t_3 > v & \ goto B_2 \\
j &:= j-1 \\
t_4 &:= 4*j \\
t_5 &:= a[t_4] \\
if \ t_5 < v & \ goto B_2 \\
if \ i+1 & \ goto B_6 \\
x &:= t_3 \\
a[t_3] &:= t_3 \\
a[t_3] &:= x \\
goto B_5
\end{align*}
\]
Copy Propagation

- Assignments of the form \( f := g \) are called copy statements (or copies)
- The idea behind copy propagation is to use \( g \) for \( f \) whenever possible after such a statement
- For example, applied to block B5 of the previous flow graph, we obtain:
  \[
  x := t3 \\
  a[t2] := t5 \\
  a[t4] := t3 \\
  \text{goto B2}
  \]
- Copy propagation often turns the copy statement into "dead code"

Dead-Code Elimination

- Dead code includes code that can never be reached and code that computes a value that never gets used
- Consider: if (debug) print …
  - It can sometimes be deduced at compile time that the value of an expression is constant
  - Then the constant can be used in place of the expression (constant folding)
  - Let's assume a previous statement assigns debug := false and value never changes
  - Then the print statement becomes unreachable and can be eliminated
- Consider the example from the previous slide
  - The value of \( x \) computed by the copy statement never gets used after the copy propagation
  - The copy statement is now dead code and can be eliminated
Loop Optimizations

- The running time of a program may be improved if we do both of the following:
  - Decrease the number of statements in an inner loop
  - Increase the number of statements in the outer loop
- Code motion moves code outside of a loop
  - For example, consider:
    ```
    while (i <= limit - 2) …
    ```
  - The result of code motion would be:
    ```
    t = limit – 2
    while (i <= t) …
    ```

Loop Optimizations

- Induction variables: variables that remain in "lock-step"
  - For example, in block B3 of previous flow graph, j and t4 are induction variables
  - Induction-variable elimination can sometimes eliminate all but one of a set of induction variables
- Reduction in strength replaces a more expensive operation with a less expensive one
  - For example, in block B3, t4 decreases by four with every iteration
  - If initialized correctly, can replace multiplication with subtraction
  - Often application of reduction in strength leads to induction-variable elimination
- Methods exist to recognize induction variables and apply appropriate transformations automatically
Loops in Control Flow Graphs

- At the intermediate language level, we lose the high-level control constructs
- Given a CFG, we want to find the loops
  - Important for optimization
- How do we do it?
  - Find a single entry node (or header), such that all paths from outside the loop to any node in the loop goes through the entry
  - This is a strongly connect region (SCR)
    - It is possible to go from any node of the SCR to any other node in the SCR
Dominators

- SCR are found by using dominators
- Node $d$ of a CFG dominates node $n$ if every path from the initial node of the CFG to $n$ goes through $d$
- Every node dominates itself and the entry node dominates all nodes in the loop

Dominator Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>everything</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>all but 1 &amp; 2</td>
</tr>
<tr>
<td>4</td>
<td>all but 1, 2, &amp; 3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7, 8, 9, 10</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
**Dominator Algorithm**

D(n₀) = {n₀}
for n ∈ N – {n₀}
D(n) = N
end for
Change = TRUE
while ( Change )
    Change = FALSE
    for n ∈ N – {n₀}
        NewDom = {n} \cup ( \cap D(p) for all pred(n) )
        if ( D(n) ≠ NewDom )
            then Change = TRUE
        end if
        D(n) = NewDom
    end for
end while

**Natural Loops**

- **Back edge**
  - Edge whose head dominates its tail

- The *natural loop* formed from the back edge n → d can be identified by finding those nodes that can reach n without going through d
  - These nodes, plus d, form the natural loop
Algorithm for Natural Loops

Input – a back edge n → d and a control flow graph
procedure insert( n )
  if ( m ∈ Loop )
    Loop = Loop ∪ {m}
    push( m, stack )
  end if
end procedure

Stack = φ
Loop = { d }
insert( n )
while ( !empty( Stack ) )
  m = pop( Stack )
  for p ∈ pred( m )
    insert( p )
  end for
end while
Natural Loops

<table>
<thead>
<tr>
<th>Back Edge</th>
<th>Natural Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 → 3</td>
<td>{3,4,5,6,7,8,10}</td>
</tr>
<tr>
<td>7 → 4</td>
<td>{4,5,6,7,8,10}</td>
</tr>
<tr>
<td>8 → 3</td>
<td>{3,4,5,6,7,8,9,10}</td>
</tr>
<tr>
<td>9 → 1</td>
<td>{1,2,3,4,5,6,7,8,9,10}</td>
</tr>
<tr>
<td>10 → 7</td>
<td>{7,8,10}</td>
</tr>
</tbody>
</table>

Another Example

D(1) = {1}
D(2) = {1,2}
D(3) = {1,2,3}
D(4) = {1,2,3,4}
D(5) = {1,2,3,4,5}
D(6) = {1,2,3,4,6}
D(7) = {1,2,7}

D(5) → 4 = {4,5}
D(6) → 2 = {2,3,4,5,6}

Dominators

Natural Loops
More on Natural Loops

- Natural loops have a useful property
  - If two loops have the same header, they are either:
    - Disjoint or
    - One is entirely contained (nested within) the other
- Thus neglecting loops with the same header we have the notion of an *inner loop*
  - One that contains no other loop

Transformations on Loops

- Several transformations require us to move statements “before the header”
  - Thus we begin treatment of a loop \( L \) by creating a new block, called the *preheader*
  - The preheader has:
    - Only the header as a successor
    - All edges that formally entered the header of \( L \) from outside \( L \), instead enter the preheader
  - Edges inside the loop are not changed
  - Initially the preheader is empty but transformation will place statements in it
Data Flow Analysis

- Pre-execution process of ascertaining and collecting information about possible run-time modification, preservation, and usage of certain quantities in a program
- Levels on which we perform data flow analysis
  - Statements (high-level or intermediate)
  - Basic blocks (intrablock)
  - Procedures (intraprocedural)
  - Program (interprocedural)
- The first two are local analysis while the last two are global analysis

Data Flow Analysis

- In this class, we are only interested in intraprocedural analysis
- We look at the following problems:
  - Reaching definitions
  - Live variables
  - Live definitions
  - Definition-use chaining
- The latter two problems can be solved from solutions to the first two problems
Data Flow Analysis

- We assume the following:
  - All relevant local data flow information is available for a particular procedure
  - Any variable aliasing is known
  - The CFG for the procedure is given
- Information can be associated with the top or the bottom of a node
- The sets of information can be conveniently represented by bit vectors

Reaching Definitions

- The problem is to determine the sets \( RDTOP(x) \) and \( RDBOT(x) \) of variable definitions that can “reach” the top and bottom of each node \( x \) in the CFB
- A definition-clear path with respect to a variable \( v \) means there is no definition of \( v \) on that path
Reaching Definitions

- A definition $d$ of a variable $v$ in a node $x$ is said to reach the top (bottom) of node $y$ iff $d$ occurs in node $x$ and there is a definition-clear path for $v$ from $d$ to the top (bottom) of node $y$.
- The following information has to be available at the bottom of each node $x$:
  - XDEFS($x$) – set of locally exposed definitions of node $x$
    - A locally exposed definition is the last definition of a variable in the node
  - PRESERVED($x$) – set of definitions preserved by node $x$
    - A definition $d$ of a variable $v$ is said to kill all definitions of the same variable that reach $d$.
    - Any definition of a variable $v$ that reaches the top of a node $x$, and there is no definition of variable $v$ in $x$ is said to be preserved by $x$.

Equations for Reaching Definitions

- $RDTOP(x) = \bigcup ( RDBOT(p) \; \forall \; p \in \text{pred}(x) )$
- $RDBOT(x) = ( RDTOP(x) \cap \text{PRESERVED}(x) ) \cup \text{XDEFS}(x)$
- If $\text{pred}(s) = \phi$, where $s$ is the start node, then $RDTOP(x) = \phi$
Algorithm for Reaching Definitions

for every block x
    RDTOP(x) = ϕ
    RDBOT(x) = XDEFS(x)
end for

Change = TRUE
while ( Change)
    Change = FALSE
    for every block x
        NewTop = ∪ RDBOT(p) ∀ p ∈ pred(x)
        if ( NewTop ≠ RDTOP(x) )
            then Change = TRUE
        end if
        RDTOP(x) = NewTop
        RDBOT(x) = ( RDTOP(x) ∩ PRESERVED(x) ) ∪ XDEFS(x)
    end for
end while

Example of Reaching Definitions

\begin{tabular}{|c|c|c|}
\hline
Block & XDEFS & PRESERVED \\
\hline
B1 & 11000 & 00000 \\
B2 & 00100 & 01011 \\
B3 & 00010 & 10100 \\
B4 & 00001 & 10100 \\
B5 & 00000 & 11111 \\
\hline
\end{tabular}
Example of Reaching Definitions

<table>
<thead>
<tr>
<th>Block</th>
<th>RDTOP</th>
<th>RDBOT</th>
<th>RDTOP</th>
<th>RDBOT</th>
<th>RDTOP</th>
<th>RDBOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>00000</td>
<td>11000</td>
<td>00100</td>
<td>11000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>00000</td>
<td>00100</td>
<td>11000</td>
<td>01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>00000</td>
<td>00010</td>
<td>01100</td>
<td>00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>00000</td>
<td>00001</td>
<td>00110</td>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>00000</td>
<td>00000</td>
<td>00111</td>
<td>00111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pass 2

<table>
<thead>
<tr>
<th>Block</th>
<th>RDTOP</th>
<th>RDBOT</th>
<th>RDTOP</th>
<th>RDBOT</th>
<th>RDTOP</th>
<th>RDBOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>01100</td>
<td>11000</td>
<td>01111</td>
<td>11000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>11111</td>
<td>01111</td>
<td>11111</td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>01111</td>
<td>00110</td>
<td>01111</td>
<td>00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>00110</td>
<td>00101</td>
<td>00110</td>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>00111</td>
<td>00111</td>
<td>00111</td>
<td>00111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pass 3

Live Variables

- Determine the sets LVTOP(x) and LVBOT(x) of variables that are live
  - A variable is live if it is referenced without modification
- A variable \( v \) in node \( y \) is said to be live at the bottom (top) of node \( x \) iff a use of \( v \) occurs in \( y \) and there is a definition-clear path for \( v \) from the use backward to the bottom (top) of node \( x \).
Live Variables

- Local information necessary:
  - $\text{XUSES}(x)$
    - Set of variable with locally exposed uses in node $x$.
    - A locally exposed use of a variable $v$ is a use of $v$ in a node $x$ that is not preceded by a definition of $v$ in $x$
  - $\text{PRESERVED}(x)$
    - As previously defined
- $\text{LVBOT}(x) = \bigcup (\text{LVTOP}(s) \forall s \in \text{succ}(x))$
- $\text{LVTOP}(x) = (\text{LVBOT}(x) \cap \text{PRESERVED}(x)) \cup \text{XUSES}(x)$
- If $\text{succ}(w) = \phi$ then $\text{LVBOT}(w) = \phi$
- We can use the same algorithm as previously given to solve the equations

Live Definitions

- A definition $d$ is live at the top of node $x$ iff
  - $d \in \text{RDTOP}(x) \cap \text{LBTOP}(x)$
- The live definition problem is different from the live variable problem because live definition gives stronger information
- This problem is useful in allocating registers
Definition-Use Chaining

- Information available for node $x$
  - $\text{RDTOP}(x)$
  - $\text{LVBOT}(x)$
  - $\text{XDEFS}(x)$
  - $\text{XUSES}(x)$
- Using both $\text{RDTOP}(x)$ and $\text{XUSES}(x)$ together, we can establish a pointer from each use in $\text{XUSES}(x)$ to the location of zero or more definitions in $\text{RDTOP}(x)$
- A similar association can be established between $\text{LVBOT}(x)$ and $\text{XDEFS}(x)$
- This double chaining is called \textit{definition-use chaining}
- Combined with other local information, we know, for a given definition, what uses might be affected by it and, for each use, what definitions can affect it

Definition-Use Chaining

- These chains are useful for:
  - Dead code elimination
  - Constant propagation
  - Error detection
Available Expressions

- An expression, such as $y+z$, is available at a point $p$ in a CFG iff every sequence of branches that the program may take to $p$ causes $y+z$ to have been computed after that last computation of $y$ or $z$.
- Thus we may eliminate the redundant computation of some expressions within each node.

Available Expressions

- Assume the following local information
  - NOTKILL($x$)
    - Set of expressions that are not killed in node $x$
    - Expression $y+z$ is killed iff the value of either $y$ or $z$ may be modified within node $x$
  - GEN($x$)
    - Set of expression generated in node $x$
    - Expression $y+z$ is generated if it is evaluated within node $x$ and neither $y$ nor $z$ is subsequently modified within node $x$
Available Expressions

- Equations:
  - $AETOP(x) = \cap (AEBOT(p) \quad \forall p \in \text{pred}(x))$
  - $AEBOT(x) = (AETOP(x) \cap \text{NOTKILL}(x)) \cup GEN(x)$
  - $AETOP(s) = \phi$ if $s$ in the initial node

Very Busy Expressions

- Let $E$ be the set of expressions defined in a CFG
- An expression is very busy at point $p$ in a CFG iff it is always used before it is killed
- Local information:
  - $\text{NOTKILL}(x)$
    - Previously defined
  - $\text{XEUSES}(x)$
    - Set of expressions that have exposed uses in node $x$
    - Those expressions with a definition-clear path from the entry of node $x$ to a use within node $x$
Very Busy Expressions

- Equations:
  - $\text{VBEBOT}(x) = \cap (\text{VBETOP}(s) \ \forall \ s \in \text{succ}(x))$
  - $\text{VBETOP}(x) = (\text{VBEBOT}(x) \cap \text{NOTKILL}(x)) \cup \text{XEUSES}(x)$
  - If $\text{succ}(w) = \phi$ then $\text{VBEBOT}(w) = \phi$

Summary

<table>
<thead>
<tr>
<th>Forward</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Use-definition chaining</td>
<td>Copy propagation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Backward</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live variable</td>
<td>Very busy expressions</td>
<td></td>
</tr>
<tr>
<td>Definition-use chaining</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS 5300 - SJAllan   Code Optimization 53

CS 5300 - SJAllan   Code Optimization 54
Summary

- **I and III**
  - Solution to equations is not unique, and we want smallest solution
  - Start with $\phi$ and iterate to solution
- **II and IV**
  - Largest solution is wanted
  - Start with universal set and iterate to solution
- **I and II**
  - Solved by iterating using depth-first search order
- **III and IV**
  - Solved by iterating in reverse depth-first search order

Transformations

- Now that we have the foundation, we will look at some transformations:
  - Common subexpressions
  - Copy propagation
  - Loop-invariant computations
  - Code motion
  - Elimination of induction variables
Common Subexpressions

- Using the information gained in available expressions, we can eliminate common subexpressions

Common Subexpressions Algorithm

for every statement s of the form \( x = y + z \) such that \( y + z \) is available at the beginning of s's block and neither y nor z is defined prior to statement s in that block

To discover the evaluations of \( y + z \) that reach s's block, we follow flow graph edges, searching backward from s's block. However we do not go through any block that evaluates \( y + z \). The last evaluation of \( y + z \) in each block encountered is an evaluation of \( y + z \) that reaches s

Create a new variable \( u \)

Replace each statement \( w = y + z \) found in the first step by:

\[
\begin{align*}
  u &= y + z \\
  w &= y \\
  \text{Replace statement } s \text{ by } x = u
\end{align*}
\]

end
Example of CSE

- $t_2 = 4 \times i$
- $t_3 = a[t_2]$
- $t_6 = 4 \times i$
- $t_7 = a[t_6]$
- $u = 4 \times i$
- $t_2 = u$
- $t_3 = a[t_2]$
- $t_6 = u$
- $t_7 = a[t_6]$

Common Subexpressions

- The algorithm does not catch all common subexpressions
  - It may be necessary to perform the algorithm a number of time in order to catch them all
Copy Propagation

- Many transformations introduce statements of the form \( x = y \)
  - Common subexpression elimination
  - Induction variable removal
- These statements can be removed many times

Copy Propagation

- We may substitute \( y \) for \( x \) in all places, provided the following conditions are met by every use of \( u \) of \( x \)
  - The ud-chain for use \( u \) consists of only \( s \)
  - On every path from \( s \) to \( u \), including paths that go through \( u \) several times (but do not go through \( s \) a second time), there are no assignments to \( y \)
- How do we solve the second condition?
  - New data flow problem
Copy Propagation

- Let CPTOP(x) by the set of copies y=z such that every path from the initial node to the beginning of x contains that statement y=z, and subsequent to the last occurrence of y=z, there are no assignments to z
- A similar definition for CPBOT(x) can be obtained
- Local sets
  - XCOPY(x)
    - Set of copies generated in node x
    - A copy s:y=z is said to be generated in block x if s occurs in x and there is no subsequent assignment to z within x
  - NOTKILL(x)
    - As previously defined

Data flow equations
- CPTOP(x) = \( \cap (CPBOT(p) \quad \forall p \in \text{pred}(x)) \)
- CPBOT(x) = (CPTOP(x) \cap NOTKILL(x)) \cup XCOPY(x)
Copy Propagation Example

```
\begin{align*}
\text{Local Information} \\
\begin{array}{|c|c|c|}
\hline
\text{Block} & \text{XCOPY} & \text{NOTKILL} \\
\hline
B1 & \{x=y\} & \phi \\
B2 & \phi & \phi \\
B3 & \{x=z\} & \phi \\
B4 & \phi & \{x=y, x=z\} \\
B4 & \phi & \{x=y, x=z\} \\
B2 & \phi & \phi \\
\hline
\end{array}
\end{align*}
```

```
\begin{align*}
\text{Final} \\
\begin{array}{|c|c|c|}
\hline
\text{Block} & \text{CPTOP} & \text{CPBOT} \\
\hline
B1 & \phi & \{x=y\} \\
B2 & \{x=y\} & \phi \\
B3 & \{x=y\} & \{x=z\} \\
B4 & \{x=z\} & \{x=z\} \\
B5 & \phi & \phi \\
\hline
\end{array}
\end{align*}
```

Copy Propagation

- No copy reaches node B5 and nothing can be optimized out
- Notice that both definitions of x “reach” B5
Copy Propagation Algorithm

for every copy s: x=y
Determine those uses of x that are reached by the definition of x, namely, x: x=y
Determine whether for every use of x found in the previous step, s is in CPTOP(n), where n is the block of this particular use, and moreover, no definition of x or y occur prior to this use of x within n
If s meets the conditions of the previous step, then remove s and replace all uses of x found in the first step by y
end

Loop Invariant Computation

- One whose value does not change as long as control stays within the loop
- If we have an assignment x = y + z and all definitions of y and z come from outside the loop, the computation is invariant
- We may have some indirection
  - One invariant computation may lead to others
Loop Invariant Computation
Algorithm

Mark “invariant” those statements whose operands are all
either constant or have all their reaching definitions outside
L
repeat
Mark “invariant” all those statements, not previously
marked, all of whose operands either are constant, have all
their reaching definitions outside L, or have exactly one
reaching definition, and that definition is a statement in L
marked invariant
until no more statements are marked “invariant”

Code Motion

- Moves statements from within the loop to the
preheader
- In order to perform code motion, the following
three conditions must be met. The conditions for
statement \( s: x=y+z \) are:
  - The block containing \( s \) dominates all exit nodes of the
    loop, where an exit of a loop is a node with a successor
    not in the loop
  - There is no other statement in the loop that assigns to \( x \).
    Again, if \( x \) is a temporary assigned only once, this
    condition is surely satisfied and need not be checked
  - No use of \( x \) in the loop is reached by any definition of \( x 
    \) other than \( s \). This condition too will be satisfied,
    normally, if \( x \) is a temporary
Code Motion Algorithm

function Valid ( s : statement ) : boolean
    if x’s block dominates all exits of L and
        x is not defined elsewhere in L and
        all uses in L of x can only be reached by the definitions of x in statement s
    then return true
    else return false
end if
end Valid

Use Loop Invariant algorithm

for every statement s defining x found in the previous step
    if Valid(s)
        then
            Move, in the order found by loop invariant algorithm, each statement s to
            the
            preheader, provided any operands of s that are defined in loop L have
            previously had
            their definition statements moved to the preheader
        end if
    end if
end for

Elimination of Induction Variables

- A variable x is called an induction variable of a loop L if
every time the variable x changed values, it is incremented
or decremented by some constant
- Many times we have more than one induction variable in a
loop and we eliminate many of the other induction
variables
- A basic induction variable, named I, is one whose only
assignments within the loop are of the form I=I+C, where C
is a constant or a name whose value does not change
within the loop
- We look for addition induction variables j that are defined
only once within L, and whose value is a linear function of
some basic induction variable i where j is defined
Induction Variable Detection

Find all basic induction variables by scanning the statements of $L$. We use the loop-invariant computation information here. Associated with each basic induction variable $i$ is the tripe $(i, 1, 0)$

Search for variables $k$ with a single assignment to $k$ within $L$ having one of the following forms:

$k = j \cdot b, k = b \cdot j, k = j/b, k = b \pm j$

where $b$ is a constant, and $j$ is an induction variable, basic or otherwise

Induction Variable Example

$$i = m - 1$$
$$j = n$$
$$t1 = 4 \cdot n$$
$$v = a[t1]$$

$$i = i + 1$$
$$t2 = 4 \cdot i$$
$$t3 = a[t2]$$

if $t3 < v$ goto B2

$$j = j - 1$$
$$t4 = r \cdot j$$
$$t5 = a[t4]$$

if $t5 > v$ goto B3

if $i \geq j$ goto B6

B1

B2

B3

B4

B5

B6

$$i = m - 1$$
$$j = n$$
$$t1 = 4 \cdot n$$
$$v = a[t1]$$

$$s2 = 4 \cdot i$$
$$s4 = 4 \cdot j$$

$$i = i + 1$$
$$s2 = s2 + 4$$
$$t2 = s2$$
$$t3 = a[t2]$$

if $t3 < v$ goto B2

$$j = j - 1$$
$$s4 = s4 - 4$$
$$t4 = s4$$
$$t5 = a[t4]$$

if $t5 > v$ goto B3

if $i \geq j$ goto B6

B2

B3

B4

B5

B6