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: *intra*procedurally, across basic blocks
  – Code motion, more complex common subexpression elimination, etc.

• *Inter*procedural 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 → 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?

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

if \( b \) is live

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

if \( b \) is not live
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
  \[ x = 2 \times 3.14 \text{ is replaced with } x = 6.28 \text{ (this often happens due to the use of symbolic constants with #define, etc.)} \]

• Commutativity
  – E.g. \( a = x \times y \) and \( b = y \times x \) have 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

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

\textit{start loop}

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

\textit{end loop}

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

But, this also applies transitively \ldots

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

• Add an instruction to \textit{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 \textit{Inv} are possible
  – Find an instruction that is not in \textit{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 \textit{Inv}
  – Add this instruction to \textit{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 $Inv$

• Why not two reaching defs, both already in $Inv$?
  
  ```
  if (…) a = 5; else a = 6;
  b = a+1;
  ```
  
  – Even though each definition of $a$ is in $Inv$, the value of $b$ is not guaranteed to be the same for each iteration

• But: not every loop-invariant computation (i.e., member of $Inv$) can be moved out of the loop
  
  – ```
    for (…) { a = 1; ... if (…) break; ... b = a+1; ... }
  ```
  
  – ```
    a = 1; b = a+1; for (…) { ... if (…) break; ... }
  ```
Conditions for the Actual Code Motion

• Given: a loop-invariant instruction from $Inv$

• 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 $a = ...$, this must be the only assignment to $a$ in the loop
  – for (...) { a = 1; ... if (...) { ... a = 2; ... } ... b = a+1; ... }
  – a = 1; for (...) { ... if (...) { ... a = 2; ... } ... b = a+1; ... }
Conditions for the Actual Code Motion

• Condition 3: to move instruction \texttt{a = ...}, every use of \texttt{a} in the loop must be reached \textbf{only} by this definition of \texttt{a}
  
  – Condition 2 ensures that there does not exist another definition of \texttt{a} in the loop; Condition 3 guarantees that even if there are definitions of \texttt{a} outside of the loop, they do not reach any use of \texttt{a} in the loop
  
  – \texttt{a = 1; for (...) \{ ... b = a+1; ... a = 2; ... \}}
  
  – \texttt{a = 1; a = 2; for (...) \{ ... b = a+1; ... \}}

• Given the subset of instructions from \texttt{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 \( x \)
  - To simplify our life for code motion, in this course we will never move instructions \( a[x] = y \) and \( x=a[y] \)
Code Transformation

• First, create a **preheader** for the loop
  
  – Original CFG

  – Modified CFG

• Next, consider all instructions in \( Inv \), in the order in which they were added to \( Inv \)
  
  – 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 `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
Old SDD for Translation

- $S \rightarrow \textbf{while (B) } S_1$
  - $\text{begin} = \text{newLabel()}$
  - $B.\text{true} = \text{newLabel()}$
  - $B.\text{false} = S.\text{next}$
  - $S_1.\text{next} = \text{begin}$
  - $S.\text{code} = \text{label(begin) } || \ B.\text{code} || \text{label}(B.\text{true}) || S_1.\text{code} || \text{gen("goto" begin)}$
  - Example: $\textbf{while (x < 0) y = 1; }$

```
begin
 L2: if (x < 0) goto L3;
goto L1;
B.true L3: y = 1;
goto L2;
L1: noop;
```
New SDD for Translation

• $S \rightarrow \textbf{while} (B) \ S_1$
  - $B.true = \text{newLabel}()$
  - $B.false = S.next$
  - $S_1.next = B.true$
  - $S.code = B.code \ |\ | \ \text{label}(B.true) \ |\ | \ S_1.code \ |\ | \ B.code$
  - Example: $\textbf{while} (x < 0) \ y = 1$;
    - if $(x < 0)$ goto L2;
    - goto L1;
  
  $B.true \ L2: \ y = 1$;
  - if $(x < 0)$ goto L2;
  - goto L1;

  $B.false \ L1: \ \text{noop};$

  Note: also need to do something similar for $\textbf{for}$ loops
Part 3: Other Global Optimizations

• 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 (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;
\[i = m - 1\]
\[j = n\]
\[t_1 = 4 \times n\]
\[v = a[t_1]\]

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

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

\[i = \max(i, j)\]
\[t_6 = 4 \times i\]
\[x = a[t_6]\]
\[t_7 = 4 \times i\]
\[t_8 = 4 \times j\]
\[t_9 = a[t_8]\]
\[a[t_7] = t_9\]
\[t_{10} = 4 \times j\]
\[a[t_{10}] = x\]
\[\text{goto } t_11\]

**Common subexpression elimination**

Local redundancy in B5:
- \(t_7 = 4 \times i\) already available in \(t_6\)
- \(t_{10} = 4 \times j\) already available in \(t_8\)

Local redundancy in B6:
- \(t_{12} = 4 \times i\) already available in \(t_{11}\)
- \(t_{15} = 4 \times n\) 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:
t6=4*i already available in t2
Can change x=a[t6] and a[t6]=t9
t8=4*j already available in t4
Can change t9=a[t8] and a[t8]=x

Global redundancy in B6:
t11=4*i already available in t2
Can change x=a[t11] and a[t11]=x
t13=4*n already available in t1
Can change t14=a[t13] and a[t13]=x
Common subexpression elimination

Global redundancy in B5:
\(x = a[t_2]\) already available in \(t_3\)
Can change \(x = a[t_2]\)
\(t_9 = a[t_4]\) already available in \(t_5\)
Can change \(t_9 = a[t_4]\)

Global redundancy in B6:
\(x = a[t_2]\) already available in \(t_3\)
Can change \(x = a[t_2]\)
\(t_{14} = a[t_1]\) not available in \(v\). Why?
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)

x=t3
t9=t5
a[t2]=t9
a[t4]=x
goto

true

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

true

B1
B2
true
B3
true
B4
true
B5
true
B6

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]

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

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]

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

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>=j with t2>=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.
Final program after all optimizations

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

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

ENTRY

ENTRY

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

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)

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

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

- \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 \textit{intersection} of \text{OUT}[m]
Worklist Algorithm (end detour)

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

Put the successor of ENTRY on \textit{worklist}

While (\textit{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 \textit{worklist}

In Reaching Definitions, we initialized IN\(n\) to the empty set; here we cannot do this, because of IN\(n\) = IN\(n\) \(\cap\) 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

```plaintext
j = N;
for ( i = 0 ; i < N ; i++ ) {
    b[i] = (a[i] + a[j]) / 2;
    j = i; } // assume j is not live here
```

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

```plaintext
j = N;
if (N>=1) {
    b[0] = (a[0] + a[N]) / 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

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

for ( i = 0 ; i < 4095 ; i +=2 ) {
  c[i] = a[i] + b[i];
  c[i+1] = a[i+1] + b[i+1];
} // unroll factor of 2

- 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

```c
for ( i = 0 ; i < N ; i++ )
    c[i] = a[i] + b[i];
for ( j = 0 ; j < N ; j++ )
    d[j] = c[j] * 2;
```

```c
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 [↓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