Formal Languages and Compilers
Lecture XI—Principles of Code Optimization

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

2017/18 – First Semester

Summary of Lecture XI

- Code Optimization
- Basic Blocks and Flow Graphs
- Sources of Optimization
  1. Common Subexpression Elimination
  2. Copy Propagation
  3. Dead-Code Elimination
  4. Constant Folding
  5. Loop Optimization

Code Optimization: Intro

- Intermediate Code undergoes various transformations—called Optimizations—to make the resulting code running faster and taking less space.
- Optimization *never* guarantees that the resulting code is the best possible.
- We will consider only *Machine-Independent Optimizations*—i.e., they don't take into consideration any property of the target machine.
- The techniques used are a combination of *Control-Flow* and *Data-Flow* analysis.
    - *Control-Flow Analysis.* Identifies loops in the flow graph of a program since such loops are usually good candidates for improvement.
    - *Data-Flow Analysis.* Collects information about the way variables are used in a program.

Criteria for Code-Improving Transformations

- The best transformations are those that yield the most benefit for the least effort.
  1. A transformation must preserve the meaning of a program. It's better to miss an opportunity to apply a transformation rather than risk changing what the program does.
  2. A transformation must, on the average, speed up a program by a measurable amount.
  3. Avoid code-optimization for programs that run occasionally or during debugging.
  4. **Remember!** Dramatic improvements are usually obtained by improving the source code: The programmer is always responsible in finding the best possible data structures and algorithms for solving a problem.

Quicksort: An Example Program

- We will use the sorting program *Quicksort* to illustrate the effects of the various optimization techniques.

```
void quicksort(m,n)
int m,n;
{
    int i,j,v,x;
    if (n <= m) return;
    i = m-1; j = n; v = a[n];    /* fragment begins here */
    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;    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Quicksort: An Example Program (Cont.)

- The following is the three-address code for a fragment of Quicksort.

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

Summary

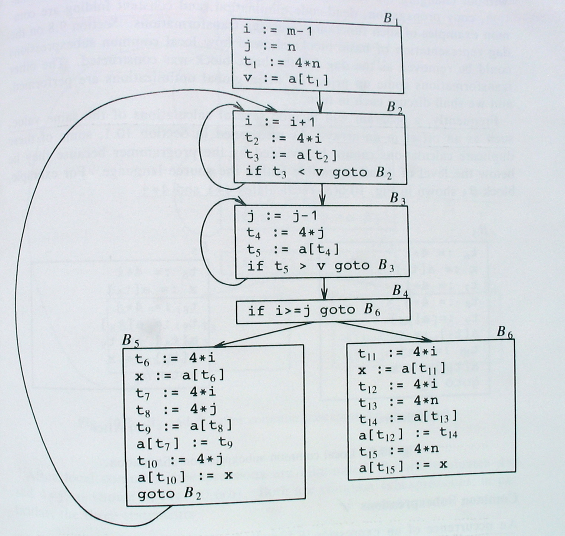
- Code Optimization
- Basic Blocks and Flow Graphs
- Sources of Optimization
    1. Common Subexpression Elimination
    2. Copy Propagation
    3. Dead-Code Elimination
    4. Constant Folding
    5. Loop Optimization

Basic Blocks and Flow Graphs

- The Machine-Independent Code-Optimization phase consists of *control-flow* and *data-flow* analysis followed by the application of transformations.
- During control-flow analysis, a program is represented as a *Flow Graph* where:
  - Nodes represent Basic Blocks: Sequence of consecutive statements in which flow-of-control enters at the beginning and leaves at the end without halt or branches;
  - Edges represent the flow of control.

Flow Graph: An Example

- Flow graph for the three-address code fragment for quicksort. Each B_i is a basic block.



The Principal Sources of Optimization

- After the *control-flow* analysis we can individuate the basic transformations as the result of *data-flow* analysis.
- We distinguish *local* transformations—involving only statements in a single basic block—from *global* transformations.
- A basic block computes a set of expressions: A number of transformations can be applied to a basic block without changing the expressions computed by the block.
  1. Common Subexpressions elimination;
  2. Copy Propagation;
  3. Dead-Code elimination;
  4. Constant Folding.

Summary

- Code Optimization
- Basic Blocks and Flow Graphs
- Sources of Optimization
  1. Common Subexpression Elimination
  2. Copy Propagation
  3. Dead-Code Elimination
  4. Constant Folding
  5. Loop Optimization

Common Subexpressions Elimination

- Frequently a program will include calculations of the same value.
- **Definition.** An occurrence of an expression E is called a Common Subexpression if E was previously computed, and the values of variables in E did not change since the previous computation.
- **Common Subexpression Elimination:** Assignments to temporary variables involving common subexpressions can be eliminated.
- **Example.** Assignments to both t_7 and t_{10} in block B_5 have common subexpressions and can be eliminated. B_5 is transformed as:

$$
\begin{aligned}
&t_6 := 4 * i \\
&x := a[t_6] \\
&t_8 := 4 * j \\
&t_9 := a[t_8] \\
&a[t_6] := t_9 \\
&a[t_8] := x \\
&\text{goto } B_2
\end{aligned}
$$

- **Example (Cont.)** After local elimination, B_5 still evaluates $4 * i$ and $4 * j$ which are *global* common subexpressions.
- $4 * j$ is evaluated in B_3 by t_4. Then, the statements

  $t_8 := 4 * j;\ \ t_9 := a[t_8];\ \ a[t_8] := x$

  can be replaced by

  $t_9 := a[t_4];\ \ a[t_4] := x$
- Now, $a[t_4]$ is also a common subexpression, computed in B_3 by t_5. Then, the statements

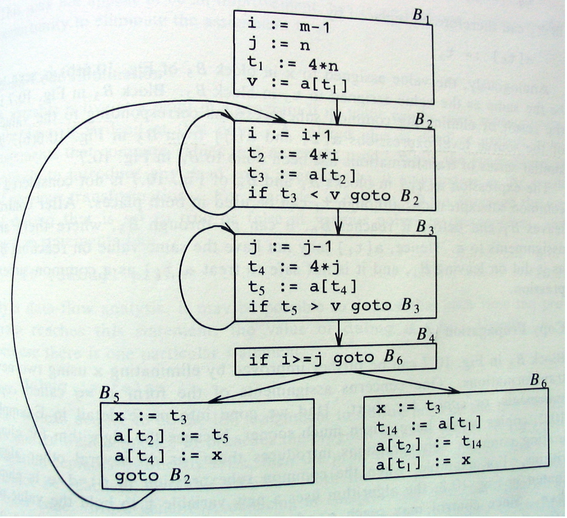
  $t_9 := a[t_4];\ \ a[t_6] := t_9$

  can be replaced by

  $a[t_6] := t_5$.
- Analogously, t_6 can be eliminated and replaced by t_2; while the value of $a[t_2]$ is the same as the value assigned to t_3 in block B_2.

Common Subexpressions Elimination (Cont.)

- **Example.** The following flow graph shows the result of eliminating both local and global common subexpressions from basic blocks B_5 and B_6.



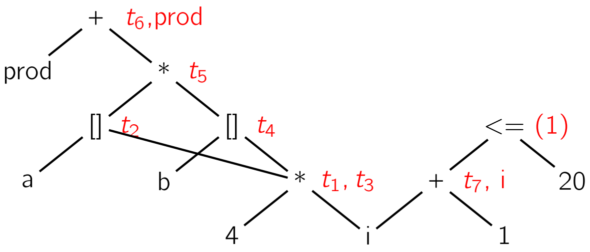
DAGs for Determining Common Subexpressions

- To individuate common subexpressions we represent a basic block as a DAG showing how expressions are re-used in a block.
- A *DAG for a Basic Block* has the following labels and nodes:
  1. Leaves contain unique identifiers, either variable names or constants.
  2. Interior nodes contain an operator symbol.
  3. Nodes can optionally be associated to a list of variables representing those variables having the value computed at the node.

DAGs for Blocks: An Example

- The following shows both a three-address code of a basic block and its associated DAG.

(1) $t_1 := 4 * i$
(2) $t_2 := a[t_1]$
(3) $t_3 := 4 * i$
(4) $t_4 := b[t_3]$
(5) $t_5 := t_2 * t_4$
(6) $t_6 := prod + t_5$
(7) $prod := t_6$
(8) $t_7 := i + 1$
(9) $i := t_7$
(10) if $i <= 20$ goto (1)



- When a node contains more temporary variables we can eliminate all but one.

Summary

- Code Optimization
- Basic Blocks and Flow Graphs
- Sources of Optimization
  1. Common Subexpression Elimination
  2. Copy Propagation
  3. Dead-Code Elimination
  4. Constant Folding
  5. Loop Optimization

Copy Propagation

- **Copy Propagation Rule**: Given the *copy statement*, `x := y`, use `y` for `x` whenever possible after the copy statement.
- Copy Propagation applied to Block B_5 yields:

$$x := t_3$$
$$a[t_2] := t_5$$
$$a[t_4] := t_3$$
$$\text{goto } B_2$$

- This transformation together with Dead-Code Elimination (see next slide) will give us the opportunity to eliminate the assignment $\mathtt{x} := \mathtt{t}_3$ altogether.

Dead-Code Elimination

- **Intuition:** A variable is *live* at a point in a program if its value can be used subsequently, otherwise it is *dead*.
- **Dead Code.** A piece of code is *dead* if data computed is never used elsewhere and can be eliminated.
- Dead-Code may appear as the result of previous transformation. Dead-Code works well together with Copy Propagation.
- **Example.** Considering the Block B_5 after Copy Propagation we can see that $\mathtt{x}$ is never reused all over the code. Thus, $\mathtt{x}$ is a dead variable and we can eliminate the assignment $\mathtt{x} := \mathtt{t}_3$ from B_5.

Constant Folding

- **Intuition:** Based on deducing at compile-time that the value of an expression (and in particular of a variable) is a constant.
- **Constant Folding** is the transformation that substitutes an expression with a constant.
- Constant Folding is useful to discover Dead-Code.
- **Example.** Consider the conditional statement: if (x) goto L.
  If, by Constant Folding, we discover that x is always false we can eliminate both the if-test and the jump to L.

Summary

- Code Optimization
- Basic Blocks and Flow Graphs
- Sources of Optimization
  1. Common Subexpression Elimination
  2. Copy Propagation
  3. Dead-Code Elimination
  4. Constant Folding
  5. Loop Optimization

Loop Optimization

- The running time of a program can be improved if we decrease the amount of instructions in an inner loop.
- Three techniques are useful:
  1. Code Motion
  2. Reduction in Strength
  3. Induction-Variable elimination

Code Motion

- If the computation of an expression is *loop-invariant* this transformation places such computation before the loop.
- **Example.** Consider the following while statement:

  ```
  while (i <= limit - 2) do
  ```

  The expression limit - 2 is loop invariant. Code motion transformation will result in:

  ```
  t := limit -2;
  while (i <= t) do
  ```

Reduction in Strength

- It is based on the replacement of a computation with a less expensive one.
- **Example.** Consider the assignment $t_4 := 4 * j$ in Block B_3.
  j is decremented by 1 each time, then $t_4 := 4 * j - 4$.
  Thus, we may replace $t_4 := 4 * j$ by $t_4 := t_4 - 4$.
  Problem: We need to initialize t_4 to $t_4 := 4 * j$ before entering the Block B_3.
  - *Result.* The substitution of a multiplication by a subtraction will speed up the resulting code.

Induction Variables

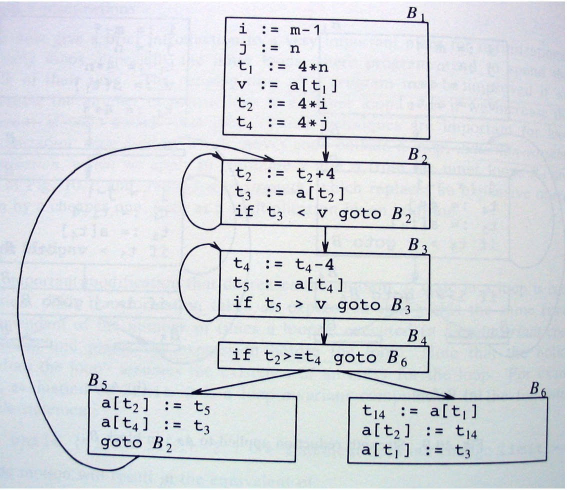
- A variable x is an Induction Variable of a loop if every time the variable x changes values, it is incremented or decremented by some constant.
- A common situation is the one in which an induction variable, say i, indexes an array, and some other induction variable, say t, is the actual offset to access the array:
  - Often we can get rid of i.
  - In general, when there are two or more Induction Variables it is possible to get rid of all but one.

Induction Variables Elimination: An Example

- **Example.** Consider the loop of Block B_3. The variables j and t_4 are Induction Variables. The same applies for variables i and t_2 in Block B_2.
- After Reduction in Strength is applied to both t_2 and t_4, the only use of i and j is to determine the test in B_4.
- Since $t_2 := 4 * i$ and $t_4 := 4 * j$, the test $i > j$ is equivalent to $t_2 > t_4$.
- After this replacement in the test, both i (in Block B_2) and j (in Block B_3) become dead-variables and can be eliminated! (see next slide for the new optimized code).

Induction Variables Elimination: An Example (Cont.)

- Flow Graph after Reduction in Strength and Induction-Variables elimination.



Summary of Lecture XI

- Code Optimization
- Basic Blocks and Flow Graphs
- Sources of Optimization
  1. Common Subexpression Elimination
  2. Copy Propagation
  3. Dead-Code Elimination
  4. Constant Folding
  5. Loop Optimization