Lecture 1
Introduction

• What would you get out of this course?
• Structure of a Compiler
• Optimization Example

What Do Compilers Do?

1. Translate one language into another
   – e.g., convert C++ into x86 object code
   – difficult for “natural” languages, but feasible for computer languages

2. Improve (i.e., “optimize”) the code
   – e.g., make the code run 3 times faster
     • or more energy efficient, more robust, etc.
   – driving force behind modern processor design

How Can the Compiler Improve Performance?

Execution time = Operation count * Machine cycles per operation

• Minimize the number of operations
  – arithmetic operations, memory accesses
• Replace expensive operations with simpler ones
  – e.g., replace 4-cycle multiplication with 1-cycle shift
• Minimize cache misses
  – both data and instruction accesses
• Perform work in parallel
  – instruction scheduling within a thread
  – parallel execution across multiple threads

What Would You Get Out Of This Course?

• Basic knowledge of existing compiler optimizations
• Hands-on experience in constructing optimizations within a fully functional research compiler
• Basic principles and theory for the development of new optimizations
II. Structure of a Compiler

- Optimizations are performed on an "intermediate form"
  - similar to a generic RISC instruction set
- Allows easy portability to multiple source languages, target machines

Ingredients in a Compiler Optimization

- **Formulate optimization problem**
  - Identify opportunities of optimization
    - applicable across many programs
    - affect key parts of the program (loops/recursions)
    - amenable to "efficient enough" algorithm
- **Representation**
  - Must abstract essential details relevant to optimization

Representation: Instructions

- Three-address code
  - A := B op C
    - LHS: name of variable e.g. x, A[t] (address of A + contents of t)
    - RHS: value
- Typical instructions
  - A := B op C
  - A := unaryop B
  - A := B
  - GOTO s
  - IF A relop B GOTO s
  - CALL f
  - RETURN
III. Optimization Example

- **Bubblesort** program that sorts an array A that is allocated in static storage:
  - an element of A requires four bytes of a byte-addressed machine
  - elements of A are numbered 1 through n (n is a variable)
  - A[j] is in location 6A+4*(j-1)

For i := n-1 DOWNTO 1 DO
  FOR j := 1 TO i DO
      temp := A[j];
      A[j] := A[j+1];
      A[j+1] := temp
    END

Example Bubblesort program that sorts an array A that is allocated in static storage:

- an element of A requires four bytes of a byte-addressed machine
- elements of A are numbered 1 through n (n is a variable)
- A[j] is in location 6A+4*(j-1)

FOR i := n-1 DOWNTO 1 DO
  FOR j := 1 TO i DO
      temp := A[j];
      A[j] := A[j+1];
      A[j+1] := temp
    END

Translation Code:

\[
\begin{align*}
  i & := n - 1 \\
  S5: \text{if } i > 1 \text{ goto } s1 \\
  & t8 := j - 1 \\
  s4: \text{if } j > 1 \text{ goto } s2 \\
  & t10 := j + 1 \\
  & t11 := A[t12] \\
  & \text{if } t3 < t7 \text{ goto } s3 \\
  & t8 := j - 1 \\
  & t9 := 4 * t8 \\
  & t10 := j + 1 \\
  & t11 := A[t12] \\
  & t12 := 4 * t11 \\
  & t13 := A[t12] \\
  & t14 := j - 1 \\
  & t15 := 4 * t14 \\
  & t16 := j - 1 \\
  & t17 := t16 - 1 \\
  & t18 := 4 * t17 \\
  & A[t18] := t18 \\
  s3: \text{if } t3 < t7 \text{ goto } s3 \\
  & t8 := j - 1 \\
  & \text{goto } s4 \\
  & s2: i := i - 1 \\
  & \text{goto } s5 \\
  s1: \\
\end{align*}
\]

Representation: a Basic Block

- **Basic block** = a sequence of 3-address statements
  - only the first statement can be reached from outside the block (no branches into middle of block)
  - all the statements are executed consecutively if the first one is (no branches out or halts except perhaps at end of block)

- We require basic blocks to be **maximal**
  - they cannot be made larger without violating the conditions

- Optimizations within a basic block are **local** optimizations

Flow Graphs

- **Nodes**: basic blocks
- **Edges**: $B_i \rightarrow B_j$ iff $B_i$ can follow $B_j$ immediately in some execution
  - Either first instruction of $B_i$ is target of a goto at end of $B_j$
  - Or, $B_i$ physically follows $B_j$ which does not end in an unconditional goto.
- The block led by first statement of the program is the **start**, or **entry** node.
Find the Basic Blocks

i := n-1
S5: if i<=1 goto s1
t := 4*t8
t10 := j+1
t11 := t10-1
t12 := 4*t11
t14 := j-1
t15 := 4*t14
t17 := t16-1
t18 := 4*t17
if t3<=t7 goto s3
j := j-1
t8 := j-1
t9 := 4*t9
t10 := j+1
t11 := t10-1
t12 := 4*t11
t13 := j+1
t14 := t13-1
t15 := 4*t14
t16 := j+1
t17 := t16-1
t18 := 4*t17
j := j+1
goto s4
S2: i := i-1
goto s5
s1:

Basic Blocks from Example

B1
B2
B3
B4
B5
B6
B7
B8

i := n-1
if i<=1 goto out
j := 1
if j>i goto B5
i := i-1
goto B2
j := j+1
goto B4
j := j+1
if t3<=t7 goto B8

Sources of Optimizations

- Algorithm optimization
- Algebraic optimization
  \[ A := B+0 \quad \Rightarrow \quad A := B \]
- Local optimizations
  - within a basic block -- across instructions
- Global optimizations
  - within a flow graph -- across basic blocks
- Interprocedural analysis
  - within a program -- across procedures (flow graphs)

Local Optimizations

- Analysis & transformation performed within a basic block
- No control flow information is considered
- Examples of local optimizations:
  - local common subexpression elimination
    analysis: same expression evaluated more than once in b.
    transformation: replace with single calculation
  - local constant folding or elimination
    analysis: expression can be evaluated at compile time
    transformation: replace by constant, compile-time value
  - dead code elimination
(Intraprocedural) Global Optimizations

- Global versions of local optimizations
  - global common subexpression elimination
  - global constant propagation
  - dead code elimination
- Loop optimizations
  - reduce code to be executed in each iteration
  - code motion
  - induction variable elimination
- Other control structures
  - Code hoisting: eliminates copies of identical code on parallel paths in a flow graph to reduce code size.
Example (After Global CSE)

B1: \( i := n-1 \)
B2: if \( i < 1 \) goto out
B3: \( j := 1 \)
B4: if \( j < 1 \) goto B5
B6: \( t1 := j-1 \)
B7: \( A[t2] := t7 \)

Example (After IV Elimination)

B1: \( i := n-1 \)
B2: if \( i < 1 \) goto out
B3: \( t2 := 0 \)
B4: if \( t6 < t19 \) goto B5
B5: \( i := i - 1 \)
B6: \( t3 := A[t2] \)
B7: \( A[t6] := t4 \)

Induction Variable Elimination

• Intuitively
  – Loop indices are induction variables (counting iterations)
  – Linear functions of the loop indices are also induction variables (for accessing arrays)

• Analysis: detection of induction variable

• Optimizations
  – strength reduction:
    • replace multiplication by additions
  – elimination of loop index:
    • replace termination by tests on other induction variables

Loop Invariant Code Motion

• Analysis
  – a computation is done within a loop and
  – result of the computation is the same as long as we keep going around the loop

• Transformation
  – move the computation outside the loop
Machine Dependent Optimizations

- Register allocation
- Instruction scheduling
- Memory hierarchy optimizations
- etc.