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
   - driving force behind modern processor design
What Do We Mean By “Optimization”?  

• Informal Definition:  
  – transform a computation to an equivalent but “better” form  
    • in what way is it equivalent?  
    • in what way is it better?  

• “Optimize” is a bit of a misnomer  
  – the result is not actually optimal
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

• Related issue: minimize object code size
  – more important on embedded systems
Other Optimization Goals Besides Performance

- Minimizing power and energy consumption

- Finding (and minimizing the impact of) software bugs
  - security vulnerabilities
  - subtle interactions between parallel threads

- Increasing reliability, fault-tolerance
Reasons for Studying Compilers

• **Compilers are important**
  - An essential programming tool
    • Improves software productivity by hiding low-level details
  - A tool for designing and evaluating computer architectures
    • Inspired RISC, VLIW machines
    • Machines’ performance measured on compiled code
  - Techniques for developing other programming tools
    • Examples: error detection tools
  - Little languages and program translations can be used to solve other problems

• **Compilers have impact: affect all programs**
Compiler Study Trains Good Developers

Excellent software engineering case study

- Optimizing compilers are hard to build
  - Input: all programs
  - Objectives:

- Methodology for solving complex real-life problems
  - Key to success: Formulate the right approximation!
    - Desired solutions are often NP-complete / undecidable
  - Where theory meets practice
    - Can't be solved by just pure hacking
      - theory aids generality and correctness
    - Can't be solved by just theory
      - experimentation validates and provides feedback to problem formulation

- Reasoning about programs, reliability & security makes you a better programmer

There are programmers, and there are tool builders...
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

• Analysis
  – Detect when it is and to apply transformation

• Code Transformation

• Experimental Evaluation (and repeat process)
Use of Mathematical Abstraction

- Design of mathematical model & algorithm
  - Generality, power, simplicity and efficiency
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 $\&A+4*(j-1)$

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

\[
i := n - 1 \\
S5: \text{ if } i < 1 \text{ goto } s1 \\
j := 1 \\
s4: \text{ if } j > i \text{ goto } s2 \\
t1 := j - 1 \\
t2 := 4 \times t1 \\
t4 := j + 1 \\
t5 := t4 - 1 \\
t6 := 4 \times t5 \\
\text{ if } t3 \leq t7 \text{ goto } s3 \\
t8 := j - 1 \\
t9 := 4 \times t8 \\
t10 := j + 1 \\
t11 := t10 - 1 \\
t12 := 4 \times t11 \\
t14 := j - 1 \\
t15 := 4 \times t14 \\
t16 := j + 1 \\
t17 := t16 - 1 \\
t18 := 4 \times t17 \\
s3: j := j + 1 \\
goto S4 \\
S2: i := i - 1 \\
goto s5 \\
s1:
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 $B_j$ can follow $B_i$ immediately in *some* execution
  - Either first instruction of $B_j$ is target of a goto at end of $B_i$
  - Or, $B_j$ physically follows $B_i$, 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

\begin{align*}
i & := n-1 \\
S5: & \quad \text{if } i<1 \text{ goto s1} \\
j & := 1 \\
s4: & \quad \text{if } j>i \text{ goto s2} \\
t1 & := j-1 \\
t2 & := 4*t1 \\
t3 & := A[t2] \ ; A[j] \\
t4 & := j+1 \\
t5 & := t4-1 \\
t6 & := 4*t5 \\
t7 & := A[t6] \ ; A[j+1] \\
\text{if } t3<=t7 & \text{ goto s3} \\
t8 & := j-1 \\
t9 & := 4*t8 \\
temp & := A[t9] \ ; A[j] \\
t10 & := j+1 \\
t11 & := t10-1 \\
t12 & := 4*t11 \\
t13 & := A[t12] \ ; A[j+1] \\
t14 & := j-1 \\
t15 & := 4*t14 \\
t16 & := j+1 \\
t17 & := t16-1 \\
t18 & := 4*t17 \\
A[t18] & := temp \ ; A[j+1] := temp \\
s3: & \quad j := j+1 \\
\text{goto } S4 \\
S2: & \quad i := i-1 \\
\text{goto } s5 \\
s1: &
\end{align*}
Basic Blocks from Example

in

B1
\[ i := n-1 \]

B2
\[ \text{if } i < 1 \text{ goto out} \]

B3
\[ j := 1 \]

B4
\[ \text{if } j > i \text{ goto B5} \]

B6
\[ t1 := j-1 \]
\[ ... \]
\[ \text{if } t3 \leq t7 \text{ goto B8} \]

B5
\[ i := i-1 \]
\[ \text{goto B2} \]

B7
\[ t8 := j-1 \]
\[ ... \]
\[ A[t18]=\text{temp} \]

B8
\[ j := j+1 \]
\[ \text{goto B4} \]

out
Sources of Optimizations

- **Algorithm optimization**

- **Algebraic optimization**
  \[ A := B + 0 \Rightarrow A := B \]

- **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
Example

\[ \begin{align*} i & := n - 1 \\
S5: \quad & \text{if } i < 1 \text{ goto } s1 \\
& j := 1 \\
s4: \quad & \text{if } j > i \text{ goto } s2 \\
t1 & := j - 1 \\
t2 & := 4 \times t1 \\
t3 & := A[t2];A[j] \\
t4 & := j + 1 \\
t5 & := t4 - 1 \\
t6 & := 4 \times t5 \\
t7 & := A[t6];A[j+1] \\
\text{if } t3 \leq t7 \text{ goto } s3 \\
t8 & := j - 1 \\
t9 & := 4 \times t8 \\
temp & := A[t9];A[j] \\
t10 & := j + 1 \\
t11 & := t10 - 1 \\
t12 & := 4 \times t11 \\
t13 & := A[t12];A[j+1] \\
t14 & := j - 1 \\
t15 & := 4 \times t14 \\
t16 & := j + 1 \\
t17 & := t16 - 1 \\
t18 & := 4 \times t17 \\
A[t18] & := \text{temp};A[j+1] := \text{temp} \\
s3: \quad & j := j + 1 \\
& \text{goto } S4 \\
S2: \quad & i := i - 1 \\
& \text{goto } s5 \\
s1: \end{align*} \]
Example

B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2]; A[j]
    t6 := 4*j
    t7 := A[t6]; A[j+1]
if t3<=t7 goto B8

B7: t8 := j-1
    t9 := 4*t8
    temp := A[t9]; temp:=A[j]
    t12 := 4*j
    t13 := A[t12]; A[j+1]
    A[t9]:= t13; A[j]:=A[j+1]
    A[t12]:=temp; A[j+1]:=temp
B8: j := j+1
    goto B4
B5: i := i-1
    goto B2
out:
(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

B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
   t2 := 4*t1
   t3 := A[t2] ;A[j]
   t6 := 4*j
   t7 := A[t6] ;A[j+1]
   if t3<=t7 goto B8
B7: t8 := j-1
   t9 := 4*t8
   t12 := 4*j
B8: j := j+1
   goto B4
B5: i := i-1
   goto B2
out:
Example (After Global CSE)

B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2];A[j]
    t6 := 4*j
    t7 := A[t6];A[j+1]
    if t3<=t7 goto B8
B7: A[t2] := t7
    A[t6] := t4
B8: j := j+1 goto B4
B5: i := i-1 goto B2
    out:
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
Example (After IV Elimination)

B1: \( i := n-1 \)

B2: if \( i<1 \) goto out

B3: \( t2 := 0 \)

\( t6 := 4 \)

B4: \( t19 := 4*I \)

if \( t6>t19 \) goto B5

B6: \( t3 := A[t2] \)

\( t7 := A[t6] ;A[j+1] \)

if \( t3<=t7 \) goto B8

B7: \( A[t2] := t7 \)

\( A[t6] := t3 \)

B8: \( t2 := t2+4 \)

\( t6 := t6+4 \)

goto B4

B5: \( i := i-1 \)

goto B2

out:
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.