Lecture 1

Introduction

I. What do you get out of this course?
II. Overview of Topics

Reference: ASU 10.1 - 10.2 (pp. 585-598)
Course Topics

- **High-level compiler structure**

  ![Diagram showing compiler structure]

  - source language 1 \( \ldots \) source language n
  - front ends
  - analysis & transformations
  - intermediate representation (IR)
  - code generation
  - machine code

- **CS 143**: front ends & code generation

- **CS 243**
  - program analysis to find out information about programs
  - transformations to change program properties
Purposes for Program Transformations

- **Optimization**
  - reduce instruction counts
  - simplify instructions
  - improve code and data locality
  - memory management
  - hide machine details: parallelization

- **Software engineering reasons**
  - memory leaks

- **Compatability**
  - binary translations
Why Study Compilers?

• **Practical need**
  • System performance: speed of system = hardware + compilers
  • Compilers are important to processor architecture development

• **Languages & compilation: heart of computing**
  • Maps a high-level abstract machine to a lower level one

• **An example of a large software program**
  • Problem solving
    • find common cases, formulate problem mathematically, develop algorithm, implement, evaluate on real data
  • Software engineering
    • build layers of abstraction (based on theory) and support with tools
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
- Understanding of the use of theory and abstraction to solve future problems
II. 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 legal and desirable to apply transformation

• Code Transformation

• Experimental Evaluation (and repeat process)
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] := A[j+1];
      A[j+1] := temp
    END
```
Translated into 3-Address Format

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

```
B1:  i := n-1
B2:  if i < 1 goto out
B3:  j := 1
B4:  if j > i goto B5
B5:  i := i - 1
goto B2
B6:  t1 := j - 1
     .........
     if t3 <= t7 goto B8
B7:  t8 := j - 1
     .........
     A[t18] = temp
B8:  j := j + 1
goto B4
```
Sources of Optimization

- 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 Optimization: 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
t14 := j-1
t15 := 4*t14
t18 := 4*j

B8: j := j+1
goto B4
B5: i := i-1
goto B2
out:
Local Optimizations

• **Examples of local optimizations**
  
  • local constant folding or elimination  
    analysis: expression can be evaluated at compile time  
    transformation: replace by constant, compile-time value
  
  • local common subexpression elimination  
    analysis: same expression evaluated more than once in b. b.  
    transformation: replace with single calculation
  
  • dead code elimination
Optimizations: (Example continued)

B1: \( i := n-1 \)  
B2: if \( i < 1 \) goto out  
B3: \( j := 1 \)  
B4: if \( j > i \) goto B5  
B6: \( t1 := j-1 \)  
  \( t2 := 4*t1 \)  
  \( t6 := 4*j \)  
  \( t7 := A[t6] ; A[j+1] \)  
  if \( t3 \leq t7 \) goto B8  
B7: \( t8 := j-1 \)  
  \( t9 := 4*t8 \)  
  \( t12 := 4*j \)  
  \( A[t9] := t13 \)  
  \( A[t12] := temp \)  
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
  - Induction variable elimination:
    Replace multiplications c*i, where i is a loop index with additions
  - Loop-invariant code motion
Optimized Example

B1: i := n-1
    t19 := 4*i
B2: if t19<4 goto out
B3: t2 := 0
    t6 := 4
B4: if t6>t19 goto B5
    t3 := A[t2]
    t7 := A[t6];A[j+1]
    if t3<=t7 goto B8
B5: t19 := t19-4
    goto B2
B6: t3 := A[t2]
    goto B2
B7: A[t2] := t7
    A[t6] := t3
B8: t2 := t2+4
    t6 := t6+4
    goto B4

• How would you write a program to perform all the optimizations?
• What are the most challenging issues?
Machine Dependent Optimizations

- Register allocation
- Instruction scheduling
Course Outline

• Introduction. Compiler optimizations.
• Basic block optimizations
• Fundamentals of data flow analysis
  • simple examples and iterative algorithms
  • theoretic foundation of data flow analysis
• Loop optimizations (scalar)
  • loop invariant expressions
  • strength reduction
• Register allocation
  • intraprocedural, interprocedural
• Code scheduling (uniprocessor) Static single assignment (SSA) program representation
• Fine grain parallelization
• Source to source restructurers for multiprocessors
• Effectiveness of compiler transformations