

CS640

Advanced Compilers

Instructor: Yutao Zhong
yzhong@cs.gmu.edu

About the Course

- **Instructor:** Dr. Yutao Zhong
 - **Email:** yzhong@cs.gmu.edu
 - **Office:** STII 419
 - **Office hour:**
 - Wednesday 4:00pm-6:00pm
 - By appointment
- **Course web site:**
 - www.cs.gmu.edu/~yzhong/cs640_s07
 - WebCT: webct41.gmu.edu

Introduction CS 640 Spring 2007 GMU 2

Course Materials

- **Main topics**
 - Compiler optimization techniques
 - Program analysis techniques
 - Parallelization, OO and functional languages support
- **Prerequisites**
 - Basic compiler construction techniques (CS540)
 - Strong programming skills
- **Resources**
 - [Engineering a Compiler](#), Cooper & Torczon
 - Dragon book, the 2nd edition
 - Papers (TBD)

Introduction CS 640 Spring 2007 GMU 3

Course Grading

- **Tentative plan**
 - **Programming assignments (45%)**
 - Tentative plan: three to four projects
 - May include non-programming part that you need to submit hard-copies
 - Late policy
 - Honor code
 - **Exams (55%)**
 - Close-book, close-notes
 - Midterm (20%) + Final (35%)
 - Make-up exam policy

Introduction CS 640 Spring 2007 GMU 4

Outline Today

- **Course logistics**
- **Motivation**
 - Why compilers/optimization compilers
 - Why study compilers/compiler optimizations
- **Review: CS540 in one lecture**
 - Compiler structure
 - Sample compiler optimizations
 - Optimization issues
 - Intermediate representations

Introduction CS 640 Spring 2007 GMU 5

Compilers

- **What is a compiler?**
 - A program that translates the input program in one language into an equivalent program in another language

"black box" view

```

graph LR
    A[Source program] --> B[Compiler]
    B --> C[Target program]
  
```

- Typically from high-level programming languages to low-level machine languages
 - Support high-level abstractions
 - Utilize low-level architectures

Introduction CS 640 Spring 2007 GMU 6

Compilers

- Common compilation tasks
 - Language translation
 - Error checking and report
 - Performance improvement
- Fundamental compilation principles
 - *The compiler must preserve the meaning of source program*
 - *The compiler must improve the source program in some discernible way*

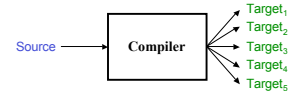
Introduction

CS 640 Spring 2007 GMU

7

Compiler Optimizations

- Optimizing compilers typically
 - Try to implement language abstractions efficiently
 - Try to utilize hardware resources efficiently
- Different targets may be generated for different compilation goals
 - Improved code speed
 - Reduced code size
 - Improved program productivity
 - Feedback to the users; debugging support
 - Shortened compilation time
 - ...



Introduction

CS 640 Spring 2007 GMU

8

Why Study Compilers?

- Compilers play a unique and critical role in software development
 - Correctness checking
 - Portability improvement
 - Performance enhancement
- Compiler study brings together
 - Data structures & algorithms
 - Formal languages
 - Computer architecture
- Compiler design and construction influence
 - Language design
 - Software engineering
 - Architecture

Introduction

CS 640 Spring 2007 GMU

9

Modern Compilers Progress

- Machines keep changing
 - New hardware features
 - Changing concerns
- Languages keep changing
 - New language features
- Applications keep changing
 - Interactive, real-time, mobile
- Desired target properties keep changing
 - Code size, running time, power consumption, security

Introduction

CS 640 Spring 2007 GMU

10

Why Study Advanced Compilers?

- An opportunity to explore compiler techniques in both breadth and depth
 - OO? Parallelization? Functional?
 - Optimizations with more details
- Compiler optimizations rely on both program analysis and transformation, which are useful in many related areas
 - Software engineering: program understanding / reverse engineering / debugging
 - Run-time support and improvement
- Open problems
 - Engineering effort: limits and issues
 - Motivate research topics

Introduction

CS 640 Spring 2007 GMU

11

Course Emphasis

- Theoretical framework
 - Algorithms and their analysis
 - “What”, “how”, “why”
- Experimentation
 - Hands-on experience
- Non-goals
 - Cover every analysis/transformation
 - Build a complete optimization compiler

Introduction

CS 640 Spring 2007 GMU

12

Outline Today

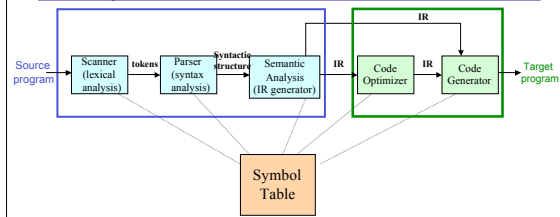
- Course logistics
- Motivation
- Review: CS540 in one lecture
 - Compiler structure
 - Sample compiler optimizations
 - Optimization issues
 - Intermediate representations

Introduction

CS 640 Spring 2007 GMU

13

Compiler Structure



- Front-end: source program to intermediate code
 - Analyze syntax and semantics of the input program
- Back-end: intermediate code to target program
 - Transform and generate the output program

Introduction

CS 640 Spring 2007 GMU

14

Lexical Analysis

- Character stream → token stream
 - Recognize “words” of language
- Theoretical problem: specify and recognize patterns in strings
 - Scanner as a practical application
 - Regular expression, finite automata
 - Tools that automatically generate scanners are commonly used

Input: index := start + step * 20
 After scanning: index := start + step * 20

identifier operator number

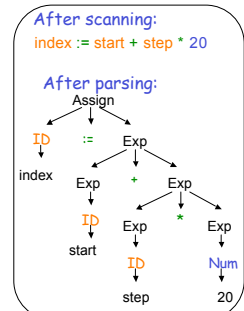
Introduction

CS 640 Spring 2007 GMU

15

Syntactical Analysis

- Token stream → syntax tree
 - Recognize “sentences” of the language
- Grammars and parsers
 - CFG
 - Parsers can be automatically generated
 - Top-down and bottom-up parsing
 - Predictive parsing
 - Driven process of compiler front-ends



Introduction

CS 640 Spring 2007 GMU

16

Semantic Analysis

- Understand/annotate meaning of the program
 - Syntax-directed translation
 - Check semantic errors
 - Inconsistent variable definitions and uses
 - Type systems
 - Collect knowledge of the input program
 - Symbol tables
 - Scopes

Introduction

CS 640 Spring 2007 GMU

17

Intermediate Code Generation

- Representation of the input program
 - Internal to the compiler
 - Encode knowledge collected during compilation
 - Varied forms and levels
 - Typically a compiler use more than one kind of IR
 - Desired properties
 - Easy to produce, manipulate, and translate into the target code

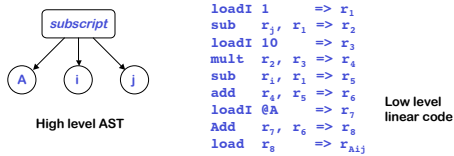
Introduction

CS 640 Spring 2007 GMU

18

Intermediate Representations

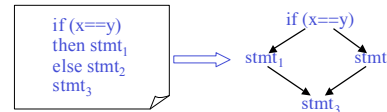
- High-level
 - Suitable to represent language abstractions
 - Syntax trees
 - Commonly used in syntax and semantic analysis
- Low-level
 - Easy to translate into target code
 - Three-address code



Introduction CS 640 Spring 2007 GMU 19

Sample IRs

- Symbol table
 - Map variable names to attributes
 - Types, scopes, procedure parameters
- Control flow graph (CFG)
 - Basic blocks
 - Loops



Introduction CS 640 Spring 2007 GMU 20

Translating Language Abstractions

- Aggregated data structure
 - Arrays, structures
 - Symbol table
 - Address calculation for references
- Control flows
 - If-then-else, for-loop, while-loop, switch-case
 - Labels and simple branches
- Procedures
 - Activation records
 - Run-time stack
- Objects
 - Object representation
 - Method dispatching

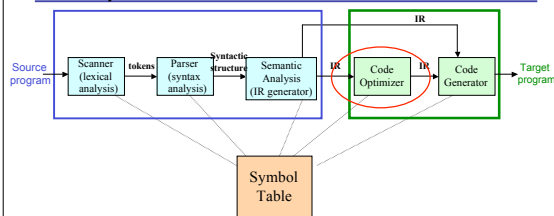
Introduction CS 640 Spring 2007 GMU 21

Code Generation

- Intermediate representation → target program
- Important issues
 - Instruction selection
 - Instruction scheduling
 - Register allocation
- Often involves machine-dependent optimizations

Introduction CS 640 Spring 2007 GMU 22

Compiler Structure



- Front-end: src program to intermediate code
 - Analyze syntax and semantics of the input program
- Back-end: intermediate code to target program
 - Transform and generate the output program

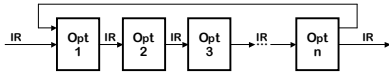
Introduction CS 640 Spring 2007 GMU 23

Code Optimizations

- Definition
 - An *optimization* is a transformation that is expected to improve program in some way
 - Program running time, memory assumption, power consumption, ect.
- Basic components
 - Code analysis: analyze the code to derive knowledge about run-time behavior
 - Code rewriting: use collected knowledge to improve the code

Introduction CS 640 Spring 2007 GMU 24

Code Optimizations



Modern optimizers are structured as a series of passes

Optimizers

- Work on some form of intermediate representations
- Multiple passes are typical
 - Each with a particular optimization task

Introduction

CS 640 Spring 2007 GMU

25

Evaluation of Code Optimizations

- Safety
 - Need to be sure the transformed code preserves the meaning of the original code
 - Generate the same output given the same input
- Profitability
 - Need to improve program in some way
 - Enabling optimizations: interaction with other transformations
- Opportunity
 - Need to know where/when to apply
 - Average case or special programs/inputs only
- Overhead
 - Must be practically computable
 - Should be justified by the profit

Introduction

CS 640 Spring 2007 GMU

26

Types of Code Optimizations

- Machine-independent optimizations
 - Eliminate redundant computation
 - Eliminate dead code
 - Perform computation at compile time if possible
 - Execute code less frequently
- Machine-dependent optimizations
 - Hide latency
 - Parallelize computation
 - Replace expensive computations with cheaper ones
 - Improve memory performance

Introduction

CS 640 Spring 2007 GMU

27

Scopes of Code Optimizations

- Peephole optimizations
 - Consider a small window of instructions
- Local optimizations
 - Consider instruction sequences within a basic block
- Intra-procedural(global) optimizations
 - Consider multiple basic blocks within a procedure
 - Need support from control flow analysis
 - Branches, loops, merging of flows
- Inter-procedural optimizations
 - Consider the whole program w/ multiple procedures
 - Need to analyze calls/returns

Introduction

CS 640 Spring 2007 GMU

28

Sample Optimizations

- Redundant loads and stores elimination


```
MOV R0, a      =>  MOV R0, a
MOV a, R0
```
- Unreachable code elimination


```
GOTO L2
x := x + 1      =< unreachable
```
- Algebraic identities


```
x := x + 0      =< can eliminate
x := x * 1
```
- Reduction in strength


```
x := x * 2      =>  x := x + x
```
- Constant folding


```
p = 2 * 3.14    =>  p = 6.28
```
- Constant propagation


```
p = 2.68
x = x * p        =>  p=2.68
                  x = x * 6.28
```

Introduction

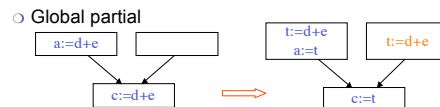
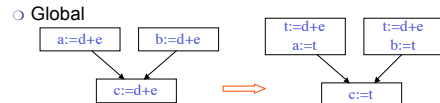
CS 640 Spring 2007 GMU

29

Sample Optimizations

- Common sub-expression elimination

Local
 $m = 2 * y * z$
 $o = 2 * y - z$ \Rightarrow $t = 2 * y$
 $m = t * z$
 $o = t - z$



Introduction

CS 640 Spring 2007 GMU

30

Sample Optimizations

Loop optimizations

Code motion

```
while (i <= limit - 2) { ... } ⇒ t := limit - 2
                               while (i <= t) { ... }
```

Loop unrolling

```
do i=1 to n by 1           do i=1 to n by 4
  a(i) = a(i)+b(i)         a(i) = a(i)+b(i)
end                         a(i+1) = a(i+1)+b(i+1)
                           a(i+2) = a(i+2)+b(i+2)
                           a(i+3) = a(i+3)+b(i+3)
                           end
                           ... //process tail part
```

Introduction

CS 640 Spring 2007 GMU

31

No Magic Bullet

Fully optimizing compiler (FOC)

- For any input program P, FOC guarantees to generate the fastest/smallest translation with the same behavior

- Impossible to construct

- Otherwise, we solve the halting problem!

References

- "Modern compiler implementation in ML" by Andrew W. Appel
- Rice's Theorem

Introduction

CS 640 Spring 2007 GMU

32

Engineering A Compiler

Practical issues

- Order and interaction of multiple optimization passes

- Balance of multiple goals

- Compilation time, code size, program running time

Compilers are engineered objects

- Thorough analysis and inference
- Careful design of internal data structures
- Careful selection of transformations
- Careful ordering of transformations

Introduction

CS 640 Spring 2007 GMU

33

Next Lecture

Topic: control flow analysis

- Basic blocks, loops, dominators

References

- EAC p.415-417, p.439-441, 9.3.2(p.457-463)
- Dragon 9.6

Introduction

CS 640 Spring 2007 GMU

34