CS380C Compilers

Instructor: Calvin Lin
lin@cs.utexas.edu
Office Hours: Mon/Wed 3:30-4:30
GDC 5.512

TA: Jia Chen
jchen@cs.utexas.edu
Office Hours: Tue 3:30-4:30
Thu 3:30-4:30
GDC 5.440

January 21, 2015

Introduction

Today’s Plan

- Motivation
  - Why study compilers?

- Let’s get started
  - Look at some sample optimizations and assorted issues

- A few administrative matters
  - Course details
Motivation

- Q: Why study compilers?

Life B.C.

- Before compilers

Machine Code

Hardware

January 21, 2015

Introduction
Liberation

Along came Backus

High-level Code

Hardware

Compilers liberate the programmer from the machine

Traditional View of Compilers

- Translate high-level language to machine code

- High-level programming languages
  - Increase programmer productivity
  - Improve program maintenance
  - Improve portability

- Low-level architectural details
  - Instruction set
  - Addressing modes
  - Registers, cache, and the rest of the memory hierarchy
  - Pipelines, instruction-level parallelism
Optimization

- Translation is not enough
  - Backus recognized the importance of obtaining good performance
- Can perform tedious optimizations that programmers won’t do

Consider Matrix Multiplication

- Obvious code
  
  ```c
  for i = 1 to n
    for j = 1 to n
      for k = 1 to n
        c[i,j] = c[i, j] + a[i, k]* b[k,j]
  ```

- Tiled code – can be significantly faster
  
  ```c
  for it = 1 to n by t
    for jt = 1 to n by t
      for kt = 1 to n by t
        for i = it to it+t-1
          for j = jt to jt+t-1
            for k = kt to kt+t-1
              c[i,j] = c[i, j] + a[i, k]* b[k,j]
  ```

Why don’t we want programmers to write this code?
Translation + Optimization

- Enable language design to flourish
  - Functional languages
  - Object oriented languages
  - ...

Compilers liberate language designers

- Logic languages

Isn’t Compilation A Solved Problem?

- “Optimization for scalar machines is a problem that was solved ten years ago”
  -- David Kuck, 1990

- Languages keep changing
  - Wacky ideas (e.g., OOP and GC) have gone mainstream

- Machines keep changing
  - New features present new problems (e.g., MMX, IA64, trace caches)
  - Changing costs lead to different concerns (e.g., loads)

- Applications keep changing
  - Interactive, real-time, mobile
Isn’t Compilation A Solved Problem? (cont)

- Values keep changing
- We used to just care about run-time performance
- Now?
  - Compile-time performance
  - Code size
  - Correctness
  - Energy consumption
  - Security
  - Fault tolerance

Value-Added Compilation

- The more we rely on software, the more we demand more of it
- Compilers can help—treat code as data
  - Analyze the code
- Correctness
- Security
Correctness and Security

- Can we check whether pointers and addresses are valid?
- Can we detect when untrusted code accesses a sensitive part of a system?
- Can we detect whether locks are used properly?
- Can we use compilers to certify that code is correct?
- Can we use compilers to verify that a given compiler transformation is correct?

Value-Added Compilation

- The more we rely on software, the more we demand more of it
- Compilers can help—treat code as data
  - Analyze the code
- Correctness
- Security
- Reliability
- Program understanding
- Program evolution
- Software testing
- Reverse engineering
- Program obfuscation
- Code compaction
- Energy efficiency

Computation important ⇒ understanding computation important
Freedom Cuts Both Ways

- Just as compilers liberate the language designer, they also liberate the computer architect
- Can we change the ISA from one generation to the next?
  - Yes, if we trust our compilers
- Enables richer design space
  - VLIW
  - IA64
  - TRIPS
  - Multicore
  - Heterogeneous multi-core
  - Reconfigurable architectures

Benefits to the Architect (cont)

- Two benefits of the compiler
  - Can simplify the hardware by shifting burden to the compiler
    - VLIW, IA64, TRIPS, software controlled caches, Cell
  - Can let the compiler inform the hardware
    - Bias bits
    - Prefetch instructions
Virtualization is a Virtue

- High-level languages provide virtualization
  - Why is virtualization good?

- We can virtualize at many levels
  - Transmeta: dynamically compile x86 to VLIW
  - GPUs rely on dynamic compilation
  - JVMs and JITs

The Point

- Compilers are a fundamental building block of modern systems

- We need to understand their power and limitations
  - Computer architects
  - Language designers
  - Software engineers
  - OS/Runtime system researchers
  - Security researchers
  - Formal methods researchers (model checking, automated theorem proving)
Plan For Today

- Motivation
  - Why study compilers?

- Let’s get started
  - Look at some sample optimizations and assorted issues

- A few administrative matters
  - Course details

Types of Optimizations

- Definition
  - An *optimization* is a transformation that is expected to improve the program in some way; often consists of *analysis* and *transformation*.
    - e.g., decreasing the running time or decreasing memory requirements

- Machine-independent optimizations
  - Eliminate redundant computation
  - Move computation to less frequently executed place
  - Specialize some general purpose code
  - Remove useless code
Types of Optimizations (cont)

- **Machine-dependent optimizations**
  - Replace a costly operation with a cheaper one
  - Replace a sequence of operations with a cheaper one
  - Hide latency
  - Improve locality
  - Reduce power consumption

- **Enabling transformations**
  - Expose opportunities for other optimizations
  - Help structure optimizations

Sample Optimizations

- **Arithmetic simplification**
  - **Constant folding**
    - *e.g.*, \( x = 8/2 \);
Sample Optimizations (cont)

- **Constant propagation**
  - *e.g.*, \( x = 3; \)
  \[ y = 4 + x; \]

Sample Optimizations (cont)

- **Common subexpression elimination (CSE)**
  - *e.g.*, \( x = a + b; \)
  \[ y = a + b; \]
Sample Optimizations (cont)

- **Dead (unused) assignment elimination**
  - *e.g.*, $x = 3$
  - ... $x$ not used...
  - $x = 4$
  - This assignment is dead

- **Dead (unreachable) code elimination**
  - *e.g.*, if (false == true) {
    - printf(“debugging...”);
  }  
  - This statement is dead

Sample Optimizations (cont)

- **Loop-invariant code motion**
  - *e.g.*, for $i = 1$ to $10$ do
    - $x = 3$
    - ...

  \[x = 3;\]
  \[for \ i = 1 \ to \ 10 \ do\]
  \[...\]
Sample Optimizations (cont)

- **Induction variable elimination**
  - *e.g.*, `for i = 1 to 10 do
    a[i] = a[i] + 1;`
    *p = *p + 1`

Sample Optimizations (cont)

- **Loop unrolling**
  - *e.g.*, `for i = 1 to 10 do
    a[i] = a[i] + 1;`
  - `for i = 1 to 10 by 2 do
    a[i] = a[i] + 1;
    a[i+1] = a[i+1] + 1;`
Is an Optimization Worthwhile?

- Criteria for evaluating optimizations
  - **Safety**: Does it preserve behavior?
  - **Profitability**: Does it actually improve the code?
  - **Opportunity**: Is it widely applicable?
  - **Cost (compilation time)**: Can it be practically performed?
  - **Cost (complexity)**: Can it be practically implemented?

Scope of Analysis/Optimizations

- **Peephole**
  - Consider a small window of instructions
  - Usually machine-specific
- **Local**
  - Consider blocks of straight line code (no control flow)
  - Simple to analyze
Scope of Analysis/Optimizations (cont)

- **Global (intraprocedural)**
  - Consider entire procedures
  - Must consider branches, loops, merging of control flow
  - Use data-flow analysis
  - Make simplifying assumptions at procedure calls

- **Whole program (interprocedural)**
  - Consider multiple procedures
  - Analysis even more complex (calls, returns)
  - Hard with separate compilation

Time of Optimization

- Compile time
- Link time
- Configuration time
- Runtime
Optimization Dimensions: A Rich Space

- Abstraction level
  - Machine-dependent, machine-independent

- Goal
  - Performance, correctness, etc
  - Enabling transformation

- Scope
  - Peephole, local, global, interprocedural

- Timing
  - Compile time, link time, configuration time, run time

January 21, 2015
Introduction

Limits of Compiler Optimizations

- Fully Optimizing Compiler (FOC)
  - FOC(P) = P_{opt}
  - P_{opt} is the smallest program with the same I/O behavior as P

- Observe
  - If program Q produces no output and never halts, FOC(Q) = L: goto L

- Aha! We’ve solved the halting problem?!

- Moral
  - Cannot build FOC
  - Can always build a better optimizing compiler

(full employment theorem for compiler writers!

January 21, 2015
Introduction

33
Optimizations Don’t Always Help

- **Common Sub-expression Elimination**

  \[
  \begin{align*}
  x &= a + b \\
  y &= a + b \\
  t &= a + b \\
  x &= t \\
  y &= t \\
  \end{align*}
  \]

  - 2 adds
  - 4 variables

  - 1 add
  - 5 variables

Optimizations Don’t Always Help (cont)

- **Fusion and Contraction**

  \[
  \begin{align*}
  \text{for } i = 1 \text{ to } n & \\
  \text{for } i = 1 \text{ to } n & \\
  C[i] &= D[i] + T[i] \\
  \text{for } i = 1 \text{ to } n & \\
  t &= A[i] + B[i] \\
  \text{for } i = 1 \text{ to } n & \\
  C[i] &= D[i] + t \\
  \end{align*}
  \]

  - \(t\) fits in a register, so no loads or stores in this loop.
  - Huge win on most machines.
  - Degrades performance on machines with hardware managed stream buffers.
Optimizations Don’t Always Help

- **Backpatching**

  In Java, the address of `foo()` is often not known until runtime (due to dynamic class loading), so the method call requires a **table lookup**.

  After the first execution of this statement, **backpatching** replaces the table lookup with a direct call to the proper function.

**Q:** How could this optimization ever hurt?

Phase Ordering Problem

- In what order should optimizations be performed?

- **Simple dependences**
  - One optimization creates opportunity for another 
    *e.g.*, copy propagation and dead code elimination

- **Cyclic dependences**
  - *e.g.*, constant folding and constant propagation

- **Adverse interactions**
  - *e.g.*, common sub-expression elimination and register allocation
  - *e.g.*, register allocation and instruction scheduling
Engineering Issues

- Building a compiler is an engineering activity

- Balance multiple goals
  - Benefit for *typical* programs
  - Complexity of implementation
  - Compilation speed

- Overall Goal
  - Identify a small set of general analyses and optimization
  - Easier said than done: just one more...

Two Approaches—Which is Better?

- Build a compiler from scratch

- Extend an existing compiler
Administrative Matters

- Turn to your syllabus

Next Time

- Reading
  - Syllabus

- Lecture
  - Undergraduate compilers in a day!