CS553 Lecture 1

Motivation

What is a compiler?
- A translator that converts a source program into a target program

What is an optimizing compiler?
- A translator that somehow improves the program

Why study compilers?
- They are specifically important:
  Compilers provide a bridge between applications and architectures
- They are generally important:
  Compilers encapsulate techniques for reasoning about programs and their behavior
- They are cool:
  First major computer application

Plan for Today

Introductions

Motivation
- Why study compilers?

Issues
- Look at some sample optimizations and assorted issues

Administrivia
- Course details

Traditional View of Compilers

Compiling down
- 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
- Pipelines
- Registers, cache, and the rest of the memory hierarchy
- Instruction-level parallelism

CS553 Lecture 1

Instructor: Michelle Strout
mstrout@cs.colostate.edu
USC 227
Office hours: decide in class

URL: http://www.cs.colostate.edu/~cs553
Isn’t Compilation A Solved Problem?

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

<table>
<thead>
<tr>
<th>Machines keep changing</th>
</tr>
</thead>
<tbody>
<tr>
<td>New features present new problems (e.g., MMX, EPIC, profiling support, multicore)</td>
</tr>
<tr>
<td>Changing costs lead to different concerns (e.g., loads)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Languages keep changing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wacky ideas (e.g., OOP and GC) have gone mainstream</td>
</tr>
</tbody>
</table>

Applications keep changing
- Interactive, real-time, mobile, secure

Some apps always want more
- More precision
- Simulate larger systems

Goals keep changing
- Correctness
- Run-time performance
- Code size
- Compile-time performance
- Power
- Security

Modern View of Compilers

Analysis and translation are useful everywhere
- Analysis and transformations can be performed at run time and link time, not just at “compile time”
- Optimization can be applied to OS as well as applications
- Analysis can be used to improve security by finding bugs
- Analysis can be used in software engineering
  - Program understanding, reverse engineering, refactoring
  - Debugging and testing
- Increased interaction between hardware and compilers can improve performance
  - Bottom line
    - Analysis and transformation play essential roles in computer systems
    - Computation important \(\Rightarrow\) understanding computation important

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 costly operation with cheaper one
- Replace sequence of operations with cheaper one
- Hide latency
- Improve locality
- Exploit machine parallelism
- Reduce power consumption

Enabling transformations
- Expose opportunities for other optimizations
- Help structure optimizations
### Sample Optimizations

#### Arithmetic simplification
- **Constant folding**
  
  Example:
  
  \[
  \begin{align*}
  x &= 8/2; \\
  x &= 4;
  \end{align*}
  \]

- **Strength reduction**
  
  Example:
  
  \[
  \begin{align*}
  x &= y * 4; \\
  x &= y \ll 2;
  \end{align*}
  \]

#### Constant propagation
- **Example:**
  
  \[
  \begin{align*}
  x &= 3; \\
  y &= 4+x; \\
  y &= 4+3; \\
  y &= 7;
  \end{align*}
  \]

#### Copy propagation
- **Examples:**
  
  \[
  \begin{align*}
  x &= z; \\
  y &= 4+x; \\
  y &= 4+z;
  \end{align*}
  \]

### Sample Optimizations (cont)

#### Common subexpression elimination (CSE)
- **Examples:**
  
  \[
  \begin{align*}
  t &= a + b; \\
  x &= t; \\
  y &= t;
  \end{align*}
  \]

#### Dead (unused) assignment elimination
- **Example:**
  
  \[
  \begin{align*}
  x &= 3; \\
  \ldots \text{x not used...} \\
  x &= 4;
  \end{align*}
  \]

#### Dead (unreachable) code elimination
- **Example:**
  
  ```
  if (false == true) {
    printf("debugging...");
  }
  ```

#### Loop-invariant code motion
- **Examples:**
  
  \[
  \begin{align*}
  x &= 3; \\
  x &= 3; \\
  \ldots
  \end{align*}
  \]

#### Induction variable elimination
- **Examples:**
  
  \[
  \begin{align*}
  a[i] &= a[i] + 1; \\
  *p &= *p + 1
  \end{align*}
  \]

#### Loop unrolling
- **Examples:**
  
  \[
  \begin{align*}
  a[i] &= a[i] + 1; \\
  a[i] &= a[i] + 1; \\
  a[i+1] &= a[i+1] + 1;
  \end{align*}
  \]

### 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

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

Limits of Compiler Optimizations

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

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

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

Moral
- Cannot build FOC
- Can always build a better optimizing compiler (full employment theorem for compiler writers!)

Optimizations Don’t Always Help

Common Subexpression 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

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

- \( t \) 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 (cont)

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?

A: The Pentium 4 has a trace cache, when any instruction is modified, the entire trace cache has to be flushed.

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 subexpression 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...

Beyond Optimization

Security and Correctness
– 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 obfuscate code?
Administrative Matters

Turn to your syllabus

Expectations

DO
– Expect to spend more time on this course than on a challenging undergraduate course.
– Write more than one draft for your project reports. Spelling mistakes will be heavily penalized. Correct grammar is also expected (for help use Word or even better the Writing Center).
– Make decisions when the project is underspecified. Describe the reasoning for your decisions in the project report.
– Read assigned reading. Much of it will take more than two readings and anything in the readings might be on the midterm or final.
– Implement your projects in small pieces, thus making them easier to debug.
– Use a debugger BEFORE coming to me about problems in your implementation.
– Come to office hours sooner rather than later.

Thinking is important and should be done frequently.

Next Time

Reading
– Chapters 1 and 2 in purple dragon book

Projects
– Take a look at project 0 (no turn in required)
– Find a partner for project 1 and get started

Lecture
– Undergrad compiler review

Concepts

Language implementation is interesting
Optimal in name only
Optimization scope
– Peephole, local, global, whole program
Optimizations
– Arithmetic simplification (constant folding, strength reduction)
– Constant/copy propagation
– Common subexpression elimination
– Dead assignment/code elimination
– Loop-invariant code motion
– Induction variable elimination
– Loop unrolling
Phase ordering problem