CIS570 Modern Programming Language Implementation

Instructor: E Christopher Lewis
Office hours: TDB
605 Levine
eclewis@cis.upenn.edu

Admin. Assistant: Cheryl Hickey
cherylh@central.cis.upenn.edu
502 Levine

URL: http://www.cis.upenn.edu/~eclewis/cis570

What is CIS570 About?

Program representation
– Can we do better than ASCII .c files?

Analysis
– How do we mechanically derive meaning and intent?
– How do we reason about programs?

Transformation
– How do we use results of analysis to make the representation “better”?

Domains
– Language implementation (including Compilation)
– Computation understanding
  – Software engineering tools (bug detectors, etc.)
Structure of CIS570

Lectures
– Participation is essential (no text book)

Reading
– 6 or 7 papers
– Read and answer discussion questions before designated class

Discussions
– Some classes will be devoted to discussion
– But there is always room for discussion

Exams
– Final perhaps?

Homework
– Answer discussion questions before class

Bottom line
– Very efficient class

Plan for Today

Motivation
– Why study compilers?

Issues
– Look at some sample optimizations and assorted issues

Administrivia
– Course details
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 (versus non-opt. comp.)

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

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
Isn’t Compilation A Solved Problem?

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

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

Machines keep changing
– New features present new problems
  (e.g., MMX, EPIC, profiling support, TM)
– Changing costs lead to different concerns (e.g., mem v. ALU ops)

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

Values 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”
– Translation can be used to improve security
– Analysis can be used in software engineering
  – Program understanding
  – Reverse engineering
– Increased interaction between hardware and compilers can improve performance
– Bottom line
  – Analysis and transformation play essential roles in computer systems
  – Computation important ⇒ 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 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., \[ x = 8/2; \] \[ \Rightarrow x = 4; \]
- Strength reduction
e.g., \[ x = y * 4; \] \[ \Rightarrow x = y \ll 2; \]

Constant propagation
- e.g., \[ x = 3; \] \[ \Rightarrow x = 3; \]
  \[ y = 4 + x; \] \[ \Rightarrow y = 4 + 3; \]
  \[ \Rightarrow y = 7; \]

Copy propagation
- e.g., \[ x = z; \] \[ \Rightarrow x = z; \]
  \[ y = 4 + x; \] \[ \Rightarrow y = 4 + z; \]

Sample Optimizations (cont)

Common subexpression elimination (CSE)
- e.g., \[ x = a + b; \]
  \[ y = a + b; \] \[ \Rightarrow t = a + b; \]
  \[ x = t; \]
  \[ y = t; \]

Dead (unused) assignment elimination
- e.g., \[ x = 3; \]
  \[ ...x \text{ not used...} \]
  \[ x = 4; \]

Dead (unreachable) code elimination
- e.g., \[ \text{if (false == true) \{ \}
  \text{printf(“debugging...”);} \]
  \[ \text{\}} \]
Sample Optimizations (cont)

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

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

Loop unrolling
- e.g., for i = 1 to 10 do
  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

<table>
<thead>
<tr>
<th>Peephole</th>
<th>Global (intraprocedural)</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Consider a small window of instructions</td>
<td>– Consider entire procedures</td>
</tr>
<tr>
<td>– Usually machine specific</td>
<td>– Must consider branches, loops, merging of control flow</td>
</tr>
<tr>
<td>– Know nothing about context</td>
<td>– Use data-flow analysis</td>
</tr>
<tr>
<td>Local</td>
<td>– Make certain assumptions at procedure calls</td>
</tr>
<tr>
<td>– Consider blocks of straight line code (no control flow)</td>
<td>Whole program (interprocedural)</td>
</tr>
<tr>
<td>– Simple to analyze</td>
<td>– Consider multiple procedures</td>
</tr>
<tr>
<td>– Know nothing about context</td>
<td>– Analysis even more complex (calls, returns)</td>
</tr>
<tr>
<td></td>
<td>– Hard with separate compilation</td>
</tr>
</tbody>
</table>

### Limits of Compiler Optimizations

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

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

\[ x = a + b \quad t = a + b \]
\[ y = a + b \quad x = t \]
\[ y = t \]

- 2 adds
- 1 add
- 4 variables
- 5 variables

Optimizations Don’t Always Help (cont)

Fusion and Contraction

for \( i = 1 \) to \( n \)
\[ C[i] = D[i] + T[i] \]

for \( i = 1 \) to \( n \)
\[ t = A[i] + B[i] \]
\[ C[i] = D[i] + t \]

\( t \) 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

```java
o.foo();
```

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?
Next Time

Reading
- “Binary Translation” by Sites et al.

Lecture
- Undergraduate compilers in a day!