CS553 Compiler Construction

Instructor: Michelle Strout
mstrout@cs.colostate.edu
USC 227
Office hours: 3-4 Monday and Wednesday

URL: http://www.cs.colostate.edu/~cs553

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

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

Machines keep changing
- New features present new problems (e.g., MMX, EPIC, profiling support)
- Changing costs lead to different concerns (e.g., loads)

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

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

Some apps always want more
- More accuracy
- 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
- 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 \Rightarrow understanding \Rightarrow 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
e.g., \( x = 8/2; \) \( \rightarrow x = 4; \)

- Strength reduction
e.g., \( x = y \times 4; \) \( \rightarrow x = y \ll 2; \)

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., \( x = a + b; \) \( y = a + b; \)
\( t = a + b; \) \( \rightarrow x = t; \)
\( \rightarrow y = t; \)

Dead (unused) assignment elimination
- e.g., \( x = 3; \)
\( ...x \) not used...
\( x = 4; \)

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

this assignment is dead

this statement is dead
Sample Optimizations (cont)

Loop-invariant code motion

- e.g., \( \text{for } i = 1 \text{ to } 10 \text{ do} \)
  \( \text{x = 3; } \)
  \( \text{... } \)
  \( \text{for } i = 1 \text{ to } 10 \text{ do} \)
  \( \text{x = 3; } \)
  \( \text{... } \)

Induction variable elimination

- e.g., \( \text{for } i = 1 \text{ to } 10 \text{ do} \)
  \( \text{for } p = &a[1] \text{ to } &a[10] \text{ do} \)
  \( a[i] = a[i] + 1; \)
  \( *p = *p + 1 \)

Loop unrolling

- e.g., \( \text{for } i = 1 \text{ to } 10 \text{ do} \)
  \( \text{for } i = 1 \text{ to } 10 \text{ by } 2 \text{ do} \)
  \( a[i] = a[i] + 1; \)
  \( a[i+1] = a[i+1] + 1; \)

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

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

Local
- Consider blocks of straight line code (no control flow)
- Simple to analyze

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

Limits of Compiler Optimizations

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

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

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

Moral
- Cannot build FOC
- Can always build a better optimizing compiler
  (\textit{full employment theorem} for compiler writers!)
Optimizations Don’t Always Help

Common Subexpression Elimination

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

\[
x = t \\
y = t
\]

2 adds 1 add
4 variables 5 variables

Optimizations Don’t Always Help (cont)

Fusion and Contraction

\[
\text{for } i = 1 \text{ to } n \\
T[i] = A[i] + B[i]
\]

\[
\text{for } i = 1 \text{ to } n \\
C[i] = D[i] + T[i]
\]

\[
\text{for } i = 1 \text{ to } n \\
t = A[i] + B[i] \\
C[i] = D[i] + t
\]

\[t\] 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.
In Java, the address of \texttt{foo()} is often not known until runtime (due to dynamic class loading), so the method call requires a \textit{table lookup}.

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

\textbf{Q:} How could this optimization ever hurt?

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

\textbf{Phase Ordering Problem}

In what order should optimizations be performed?

\textbf{Simple dependences}
- One optimization creates opportunity for another
  \textit{e.g.}, copy propagation and dead code elimination

\textbf{Cyclic dependences}
- \textit{e.g.}, constant folding and constant propagation

\textbf{Adverse interactions}
- \textit{e.g.}, common subexpression elimination and register allocation
  \textit{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

Next Time

Reading
− Intro material in Muchnick and in Bison manual

Lecture
− Scanning and parsing 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