CSE 501:
Implementation of Programming Languages

Main focus: program analysis and transformation
- how to represent programs?
- how to analyze programs? what to analyze?
- how to transform programs? what transformations to apply?
Study imperative, functional, and object-oriented languages

Official prerequisites:
- CSE 401 or equivalent
- CSE 505 or equivalent

Reading:
Appel’s “Modern Compiler Implementation”
+ ~20 papers from literature
“Compilers: Principles, Techniques, & Tools”, a.k.a. the Dragon Book, as a reference

Coursework:
- periodic homework assignments
- major course project
- midterm, final

Course outline

Models of compilation/analysis

Standard optimizing transformations

Basic representations and analyses
Fancier representations and analyses

Interprocedural representations, analyses, and transformations
- for imperative, functional, and OO languages

Run-time system issues
- garbage collection
- compiling dynamic dispatch, first-class functions, ...

Dynamic (JIT) compilation

Other program analysis frameworks and tools
- model checking, constraints, best-effort “bug finders”

Why study compilers?

Meeting area of programming languages, architectures
- capabilities of compilers greatly influence design of these others

Program representation, analysis, and transformation
is widely useful beyond pure compilation
- software engineering tools
- DB query optimizers, programmable graphics renderers
  (domain-specific languages and optimizers)
- safety/security checking of code,
  e.g. in programmable/extensible systems, networks, databases

Cool theoretical aspects, too
- lattice domains, graph algorithms, computability/complexity

Goals for language implementation

Correctness

Efficiency
- of: time, data space, code space
- at: compile-time, run-time

Support expressive, safe language features
- first-class, higher-order functions
- method dispatching
- exceptions, continuations
- reflection, dynamic code loading
- bounds-checked arrays, ...
- garbage collection
- ...

Support desirable programming environment features
- fast turnaround
- separate compilation, shared libraries
- source-level debugging
- profiling
- ...

Craig Chambers 1 CSE 501

Craig Chambers 2 CSE 501

Craig Chambers 4 CSE 501
**Standard compiler organization**

- **Analysis of input program (front-end)**
  - Character stream
  - Lexical Analysis
  - Token stream
  - Syntactic Analysis
  - Abstract syntax tree
  - Semantic Analysis
  - Interpreted
  - Intermediate Code Generation
  - Intermediate form

- **Synthesis of output program (back-end)**
  - Intermediate form
  - Optimization
  - Code Generation
  - Target language

**Mixing front-ends and back-ends**

Define intermediate language (e.g. Java bytecode, MSIL, SUIF, WIL, C, C++, ...)

Compile multiple languages into it
- each such compiler may not be much more than a front-end

Compile to multiple targets from it
- may not be much more than back-end

Or, interpret/execute it directly
Or, perform other analyses of it

Advantages:
- reuse of front-ends and back-ends
- portable “compiled” code

BUT: design of portable intermediate language is hard
- how universal?
  - across input language models? target machine models?
- high-level or low-level?

**Key questions**

- How are programs represented in the compiler?
- How are analyses organized/structured?
  - Over what region of the program are analyses performed?
  - What analysis algorithms are used?
- What kinds of optimizations can be performed?
  - Which are profitable in practice?
  - How should analyses/optimizations be sequenced/combined?

- How best to compile in face of:
  - pointers, arrays
  - first-class functions
  - inheritance & message passing
  - parallel target machines

- Other issues:
  - speeding compilation
  - making compilers portable, table-driven
  - supporting tools like debuggers, profilers, garbage collectors

**Overview of optimizations**

First **analyze** program to learn things about it
Then **transform** the program based on info
Repeat...

Requirement: don’t change the semantics!
- transform input program into
  - semantically equivalent but better output program

Analysis determines when transformations are:
- legal
- profitable

Caveat: “optimize” a misnomer
- result is almost never optimal
- sometimes slow down some programs on some inputs
  - (although hope to speed up most programs on most inputs)
Semantics

Exactly what are the semantics that are to be preserved?
Subtleties:
• evaluation order
• arithmetic properties like associativity, commutativity
• behavior in "error" cases

Some languages very precise
• programmers always know what they’re getting

Others weaker
• allow better performance (but how much?)

Semantics selected by compiler option?

Scope of analysis

Peephole: across a small number of "adjacent" instructions
[adjacent in space or time]
• trivial analysis

Local: within a basic block
• simple, fast analysis

Intraprocedural (a.k.a. global):
across basic blocks, within a procedure
• analysis more complex: branches, merges, loops

Interprocedural:
across procedures, within a whole program
• analysis even more complex: calls, returns
• hard with separate compilation

Whole-program:
analysis examines whole program in order to prove safety

A tour of common optimizations/transformations

arithmetic simplifications:
• constant folding
  \[ x := 3 + 4 \Rightarrow x := 7 \]
• strength reduction
  \[ x := y \cdot 4 \Rightarrow x := y << 2 \]

constant propagation
\[ x := 5 \Rightarrow x := 5 \Rightarrow x := 5 \]
\[ y := x + 2 \quad y := 5 + 2 \quad y := 7 \]

integer range analysis
• fold comparisons based on range analysis
• eliminate unreachable code
  \[
  \text{for(index = 0; index < 10; index ++) \{}
  \text{if index >= 10 goto \_error}
  \text{a[index] := 0}
  \text{\}}
  \]
• more generally, symbolic assertion analysis

common subexpression elimination (CSE)
\[ x := a + b \Rightarrow x := a + b \]
\[ \ldots \]
\[ y := a + b \quad y := x \]
• can also eliminate redundant memory references, branch tests

partial redundancy elimination (PRE)
• like CSE, but with earlier expression only available along
  subset of possible paths
  \[
  \text{if ... then} \Rightarrow \text{if ... then}
  \quad \ldots
  \quad \ldots
  \quad x := a + b \quad t := a + b; x := t
  \text{end} \quad \text{else} \quad t := a + b \quad \text{end}
  \quad \ldots
  \quad \ldots
  \quad y := a + b \quad y := t
  \]

copy propagation
\[
x := y \quad \Rightarrow \quad x := y
\]
\[
w := w + x \quad \Rightarrow \quad w := w + y
\]

dead (unused) assignment elimination
\[
x := y \quad \Rightarrow \quad x := y
\]
\[
\ldots \quad \text{// no use of } x
\]
\[
x := 6
\]

• a common clean-up after other optimizations:
\[
x := y \quad \Rightarrow \quad x := y \quad \Rightarrow \quad x := y
\]
\[
w := w + x \quad w := w + y \quad \Rightarrow \quad w := w + y
\]
\[
\ldots \quad \text{// no use of } x
\]

partial dead assignment elimination
• like DAE, except assignment only used on some later paths

dead (unreachable) code elimination
\[
\text{if false goto } \text{else}
\]
\[
\ldots
\]
\[
goto \_\text{done}
\]
\[
\_\text{else:}
\]
\[
\_\_\_\_\_\_\_\_
\]
\[
\_\text{done:}
\]
• another common clean-up after other optimizations

pointer/alias analysis
\[
p := &x \quad \Rightarrow \quad p := &x \quad \Rightarrow \quad p := &x
\]
\[
*p := 5 \quad \Rightarrow \quad *p := 5 \quad \Rightarrow \quad *p := 5
\]
\[
y := x + 1 \quad y := 5 + 1 \quad y := 6
\]
\[
x := 5
\]
\[
*p := 3
\]
\[
y := x + 1 \quad \Rightarrow \quad ???
\]

• augments lots of other optimizations/analyses

loop-invariant code motion
\[
\text{for } j := 1 \text{ to } 10 \quad \Rightarrow \quad \text{for } j := 1 \text{ to } 10
\]
\[
\text{for } i := 1 \text{ to } 10 \quad \Rightarrow \quad \text{for } i := 1 \text{ to } 10
\]
\[
t := b[j]
\]
\[
a[i] := a[i] + b[j]
\]
\[
\text{for } i := 1 \text{ to } 10
\]
\[
a[i] := a[i] + t
\]

induction variable elimination
\[
\text{for } i := 1 \text{ to } 10 \quad \Rightarrow \quad \text{for } p := &a[10] \text{ to } &a[10]
\]
\[
a[i] := a[i] + 1 \quad \Rightarrow \quad *p := *p + 1
\]
• a[i] is several instructions, *p is one

loop unrolling
\[
\text{for } i := 1 \text{ to } N \quad \Rightarrow \quad \text{for } i := 1 \text{ to } N \text{ by } 4
\]
\[
a[i] := a[i] + 1
\]
\[
a[i+1] := a[i+1] + 1
\]
\[
a[i+2] := a[i+2] + 1
\]
\[
a[i+3] := a[i+3] + 1
\]

parallelization
\[
\text{for } i := 1 \text{ to } 1000 \quad \Rightarrow \quad \text{forall } i := 1 \text{ to } 1000
\]
\[
a[i] := a[i] + 1
\]
• loop interchange, skewing, reversal, ...

blocking/tiling
• restructuring loops for better data cache locality
\[
\text{for } i := 1 \text{ to } 1000
\]
\[
\text{for } j := 1 \text{ to } 1000
\]
\[
c[i,j] := a[i,k] \ast b[k,j]
\]
\[
\Rightarrow
\]
\[
\text{for } i := 1 \text{ to } 1000 \text{ by TILESIZEx}
\]
\[
\text{for } j := 1 \text{ to } 1000 \text{ by TILESIZEy}
\]
\[
\text{for } k := 1 \text{ to } 1000
\]
\[
\text{for } i’ := \text{i to i+TILESIZEx}
\]
\[
\text{for } j’ := \text{j to j+TILESIZEy}
\]
\[
c[i’,j’] := a[i’,k] \ast b[k,j’]
\]
inlining
\[
l := \ldots \quad \Rightarrow \quad l := \ldots \quad \Rightarrow \quad l := \ldots
\]
\[
w := 4 \quad \quad w := 4 \quad \quad w := 4
\]
a := area(l, w) \quad a := l \cdot w \quad a := l \ll 2
• lots of “silly” optimizations become important after inlining

interprocedural constant propagation, alias analysis, etc.

static binding of dynamic calls
• in imperative languages, for call of a function pointer:
  if can compute unique target of pointer, can replace with direct call
• in functional languages, for call of a computed function:
  if can compute unique value of function expression, can replace with direct call
• in OO languages, for dynamically dispatched message:
  if can deduce class of receiver, can replace with direct call
• other possible optimizations even if several possible targets

procedure specialization

register allocation

instruction selection
\[
p1 := p + 4 \quad \Rightarrow \quad ld \ %g3, [\%g1 + 4]
\]
x := *p1
• particularly important on CISCs

instruction scheduling
\[
ld \ %g2, [\%g1 + 0] \quad \Rightarrow \quad ld \ %g2, [\%g1 + 0]
\]
add \ %g3, %g2, 1 \quad \quad ld \ %g5, [\%g1 + 4]
\]
add \ %g4, %g2, 1 \quad add \ %g4, %g5, 1
• particularly important with instructions that have delayed results, and on wide-issue machines
• vs. dynamically scheduled machines?

Optimization themes

Don’t compute it if you don’t have to
• dead assignment elimination

Compute it at compile-time if you can
• constant folding, loop unrolling, inlining

Compute it as few times as possible
• CSE, PRE, PDE, loop-invariant code motion

Compute it as cheaply as possible
• strength reduction, induction var. elimination, parallelization, register allocation, scheduling

Enable other optimizations
• constant & copy propagation, pointer analysis

Compute it with as little code space as possible
• dead code elimination

The phase ordering problem

Typically, want to perform a number of optimizations;
in what order should the transformations be performed?

some optimizations create opportunities for other optimizations
⇒ order optimizations using this dependence
• some optimizations simplified
  if can assume another opt will run later & “clean up”

but what about cyclic dependences?
• e.g. constant folding ⇒ constant propagation

what about adverse interactions?
• e.g.
  common subexpression elimination ⇒ register allocation
• e.g.
  register allocation ⇒ instruction scheduling
Compilation models

Separate compilation
- compile source files independently
- trivial link, load, run stages
  + quick recompilation after program changes
  - poor interprocedural optimization

Link-time compilation
- delay (bulk of) compilation until link-time
  + allow interprocedural & whole-program optimizations
  - quick recompilation?
  - shared precompiled libraries?
  - dynamic loading?
Examples: Vortex, some research optimizers/parallelizers, ...

Run-time compilation (a.k.a. dynamic, just-in-time compilation)
- delay (bulk of) compilation until run-time
- can perform whole-program optimizations
- can perform opts based on run-time program state, execution environment
  + best optimization potential
  + can handle run-time changes/extensions to the program
  - severe pressure to limit run-time compilation overhead
Examples: Java/.NET JITs, Dynamo, FX-32, Transmeta

Selective run-time compilation
- choose what part of compilation to delay till run-time
  + can balance compile-time/benefit trade-offs
Example: DyC

Hybrids of all the above
- spread compilation arbitrarily across stages
  + all the advantages, and none of the disadvantages!!
Example: Whirlwind (future)

Engineering

Building a compiler is an engineering activity
- balance
  complexity of implementation,
  speed-up of “typical” programs,
  compilation speed,
  ...

Near infinite number of special cases for optimization
- can’t implement them all

Good compiler design, like good language design, seeks
- small set of powerful, general analyses and transformations,
- to minimize implementation complexity while maximizing effectiveness
- reality isn’t always this pure...