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

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

**Why study compilers?**

Meeting area of programming languages, architecture
- capabilities of compilers greatly influences design of these others

Program representation and analysis is widely useful
- software engineering tools
- DB query optimizers
- programmable graphics renderers
- safety checking of code, e.g. in programmable/extensible systems, networks, databases

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

Opportunity for AI?

**Goals for language implementation**

Correctness

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

Support expressive language features
- first-class, higher-order functions
- dynamic dispatching
- exceptions, continuations
- reflection, dynamic code loading
- ...

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

**Course outline**

Models of compilation

Standard transformations

Basic representations and analyses
Fancier representations and analyses

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

Representations, analyses, and transformations for parallel machines

Compiler back-end issues
- register allocation
- instruction scheduling

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

Analysis of input program (front-end)

- Lexical Analysis
  - character stream

- Syntactic Analysis
  - token stream

- Semantic Analysis
  - abstract syntax tree

- Intermediate form

Synthesis of output program (back-end)

- Optimization

- Code Generation

- target language

- Intermediate form

Interpreters

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 collect’rs

Compiling to a portable intermediate language

Define “portable” intermediate language
(e.g. Java bytecode, MSIL, SUIF, WIL, 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

Maybe interpret/execute directly

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

Design of portable intermediate language is hard
  - how universal?
    - across input language models? target machine models?
  - fast interpretation and simple compilation at odds

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
  - 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 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
• sometimes useful
• more useful for higher-level languages
• hard with separate compilation

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

Catalog of optimizations/transformations

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

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

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

common subexpression elimination (CSE)
\[x := a + b \Rightarrow x := a + b\]
\[\ldots \Rightarrow \ldots\]
\[y := a + b \Rightarrow y := x\]
• 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
\[if \ldots \text{then} \Rightarrow if \ldots \text{then}\]
\[\ldots \Rightarrow \ldots\]
\[x := a + b \Rightarrow t := a + b; x := t\]
\[\text{end} \Rightarrow \text{else} t := a + b \text{end}\]
\[\ldots \Rightarrow \ldots\]
\[y := a + b \Rightarrow y := x\]

pointer/alias analysis
\[p := \&x \Rightarrow p := \&x \Rightarrow p := \&x\]
\[p := 5 \Rightarrow p := 5 \Rightarrow p := 5\]
\[y := x + 1 \Rightarrow y := 5 + 1 \Rightarrow y := 6\]
dead (unused) assignment elimination
\[ x := y \times z \]
... // no use of x
x := 6

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

dead (unreachable) code elimination
if false goto else
...
goto _done
else:
...
_done:

integer range analysis
• fold comparisons based on range analysis
• eliminate unreachable code
for(index = 0; index < 10; index ++)
    if index >= 10 goto _error
    a[index] := 0

loop optimizations
loop-invariant code motion
for j := 1 to 10
    for i := 1 to 10
        t := b[j]
        a[i] := a[i] + b[j]
    for i := 1 to 10
        a[i] := a[i] + t

induction variable elimination
for i := 1 to 10
    for p := a[i] + 1
        p := p + 1
    a[i] := a[i] is several instructions, *p is one

loop unrolling
for i := 1 to N
    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
for i := 1 to 1000
    a[i] := a[i] + 1
loop interchange, skewing, reversal, ...

blocking/tiling
• restructuring loops for better data cache locality

Call optimizations
inlining
l := ...
\[ l := \ldots \]
w := 4
\[ w := 4 \]
a := area(l, w)
\[ a := 1 \times w \]
a := 1 \times 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
• more generally, partial evaluation
Machine-dependent optimizations

register allocation

instruction selection

\[ p1 := p + 4 \quad \Rightarrow \quad \text{ld } \%g3, [\%g1 + 4] \]
\[ x := *p1 \]

• particularly important on CISCs

instruction scheduling

\[ \text{ld } \%g2, [\%g1 + 0] \quad \Rightarrow \quad \text{ld } \%g2, [\%g1 + 0] \]
\[ \text{add } \%g3, \%g2, 1 \quad \text{ld } \%g5, [\%g1 + 4] \]
\[ \text{ld } \%g2, [\%g1 + 4] \quad \text{add } \%g3, \%g2, 1 \]
\[ \text{add } \%g4, \%g2, 1 \quad \text{add } \%g4, \%g5, 1 \]

• particularly important with instructions that have delayed results, and on wide-issue machines
• vs. dynamically scheduled machines?

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 dependencies?
• 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 recomilation after program changes
  – poor interprocedural optimization

Link-time compilation
• delay bulk of compilation until link-time
• then perform whole-program optimizations
+ allow interprocedural & whole-program optimizations
  – quick recomilation? shared precompiled libraries?
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 + optimizations 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 JITs, Dynamo, FX-32, Transmeta

Selective run-time compilation
• choose what part of compilation to delay to run-time
+ can balance compile-time/benefit trade-offs
Examples: DyC, ...

Hybrids of all the above
• spread compilation arbitrarily across stages
+ all the advantages, and none of the disadvantages!!
Example: Whirlwind
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 be identified
- 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...