CSE 501:
Implementation of Programming Languages

Main focus: **program analysis and transformation**
- how to represent programs?
- how to analyze programs? what analyses to perform?
- how to transform programs? what transformations to apply?

Applications to compilers and software engineering tools
Applied to imperative, functional, and object-oriented languages
Advanced language runtime systems

Readings:
- 12 papers from literature
- Suggested reference books:
  - Cooper & Torczon’s “Engineering a Compiler”
  - Appel’s “Modern Compiler Implementation”
  - “Compilers: Principles, Techniques, & Tools” a.k.a. Dragon Book

Coursework:
- periodic homework assignments (~2-4)
- course project assignments (~2-3)
- midterm(?), final

Course outline

Models of compilation/analysis

Tour of standard optimizing transformations

Basic program representations and analyses
Fancier program 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

Why study compilers?

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

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/extendible systems, networks, databases

Increasing applicability of other domains to compilers
  - AI techniques to guide optimizers through search space

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

Goals for compilers

Be correct

Be efficient
  - of: time, data space, code space
  - at: compile-time, run-time

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

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

Be implementable, maintainable, evolvable, ...
**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

**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, Whirlwind (now), some other research optimizers/parallelizers, ...

**Standard compiler organization**

Analysis of input program (front-end)
- Character stream
- Lexical Analysis
- Token stream
- Syntactic Analysis
- Abstract syntax tree
- Semantic Analysis
- Annotated AST

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

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 JlTs, 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)
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 can make closed-world assumptions

A tour of common optimizations/transformations

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

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

integer range analysis
- fold comparisons based on range analysis
- eliminate unreachable code
  ```
  for(index = 0; index < 10; index++) {
    if(index > 10 goto proceed
    a[index] := 0
  }
  ```
- more generally, symbolic assertion analysis
common subexpression elimination (CSE)

\[ x := a + b \Rightarrow x := a + b \]
\[ ... \]
\[ y := a + b \quad y := x \]

- 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

\[ \text{if} \ ... \ \text{then} \Rightarrow \text{if} \ ... \ \text{then} \]
\[ ... \]
\[ x := a + b \quad t := a + b; \; x := t \]
\[ \text{end} \quad \text{else} \; t := a + b \; \text{end} \]
\[ ... \]
\[ y := a + b \quad y := t \]

copy propagation

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

dead (unused) assignment elimination

\[ x := y \]
\[ \quad \Rightarrow \quad x := y \]
\[ w := w + x \quad w := w + y \]
\[ \quad \Rightarrow \quad w := w + y \]
\[ ... \quad \Rightarrow \quad \text{no use of } x \]
\[ x := 6 \]

- a common clean-up after other optimizations:

\[ x := y \]
\[ w := w + x \quad w := w + y \]
\[ \quad \Rightarrow \quad w := w + y \]
\[ ... \quad \Rightarrow \quad \text{no use of } x \]

partial dead assignment elimination

- like DAE, except assignment only used on some later paths

dead (unreachable) code elimination

\[ \text{if} \; \text{false-goto}\_\text{else} \]
\[ ... \]
\[ \text{goto} \; \text{_done} \]
\[ \_\text{done:} \]

- another common clean-up after other optimizations

pointer/alias analysis

\[ p := 4x \quad \Rightarrow \quad p := 4x \quad \Rightarrow \quad p := 4x \]
\[ *p := 5 \quad *p := 5 \quad *p := 5 \]
\[ y := x + 1 \quad y := 5 + 1 \quad y := 6 \]
\[ x := 5 \]
\[ *p := 3 \]
\[ y := x + 1 \Rightarrow \quad ??? \]

- augments lots of other optimizations/analyses

loop-invariant code motion

\[ \text{for} \; j := 1 \; \text{to} \; N \quad \Rightarrow \quad \text{for} \; j := 1 \; \text{to} \; N \]
\[ \text{for} \; i := 1 \; \text{to} \; N \quad t := b[j] \]
\[ a[i] := a[i] + b[j] \quad \text{for} \; i := 1 \; \text{to} \; N \]
\[ a[i] := a[i] + t \]

induction variable elimination

\[ \text{for} \; i := 1 \; \text{to} \; N \quad \Rightarrow \quad \text{for} \; p := 4a[p] \; \text{to} \; 4a[N] \]
\[ a[i] := a[i] + 1 \quad *p := *p + 1 \]

- \( a[i] \) is several instructions, *p is one

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

loop peeling,...

parallellization
\[
\text{for } i := 1 \text{ to } 1000 \Rightarrow \text{forall } i := 1 \text{ to } 1000
\]
\[
a[i] := a[i] + 1 \quad a[i] := a[i] + 1
\]

loop interchange, skewing, reversal,...

blocking/tiling: restructuring loops for better cache locality
\[
\text{for } i := 1 \text{ to } 1000
\]
\[
\text{for } j := 1 \text{ to } 1000
\]
\[
\text{for } k := 1 \text{ to } 1000
\]
\[
c[i,j] := a[i,k] * b[k,j]
\]
\[
\Rightarrow
\]
\[
\text{for } i := 1 \text{ to } 1000 \text{ by TILESIZE}
\]
\[
\text{for } j := 1 \text{ to } 1000 \text{ by TILESIZE}
\]
\[
\text{for } k := 1 \text{ to } 1000
\]
\[
\text{for } i' := i \text{ to } i+\text{TILESIZE}
\]
\[
\text{for } j' := j \text{ to } j+j\text{TILESIZE}
\]
\[
c[i',j'] := a[i',k] * b[k,j']
\]

Inlining
\[
h := \ldots \Rightarrow h := \ldots \Rightarrow h := \ldots
\]
\[
w := 4 \quad w := 4
\]
\[
a := \text{area}(h,w) \quad a := h * w \quad a := h << 2
\]

- lots of "silly" optimizations become important after inlining

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 callees

Procedure specialization

Register allocation

Instruction selection
\[
\text{pl} := p + 4 \Rightarrow \text{ld } %g3, [\%g1 + 4]
\]
\[
x := \ast \text{pl}
\]

- particularly important on CISCs

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

- particularly important for instructions with delayed results, and on wide-issue machines
- less important on 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, iteration 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 dependencies?
- e.g. constant folding ⇔ constant propagation

what about adverse interactions?
- e.g.
  - common subexpression elimination ⇔ register allocation
  - e.g.
    - register allocation ⇔ instruction scheduling

---

**Option 1: high-level syntax-based representation**

Represent source-level control structures & expressions directly

**Examples**

- (Attributed) AST
- Lisp S-expressions
- extended lambda calculus

**Source:**

```plaintext```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

**AST:**

```
for
  i 1 10
  =
    a i
    [ ]
    [ ]
    * 5
    b l
```
Option 2: low-level representation

Translate input programs into low-level primitive chunks, often close to the target machine.

Examples
- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (RTLs)

Standard RTL operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>RTL code</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td><code>x := y;</code></td>
</tr>
<tr>
<td>unary op</td>
<td><code>x := op y;</code></td>
</tr>
<tr>
<td>binary op</td>
<td><code>x := y op z;</code></td>
</tr>
<tr>
<td>address-of</td>
<td><code>p := &amp;y;</code></td>
</tr>
<tr>
<td>load</td>
<td><code>x := *(p + o);</code></td>
</tr>
<tr>
<td>store</td>
<td><code>*(p + o) := x;</code></td>
</tr>
<tr>
<td>call</td>
<td><code>x := f(...);</code></td>
</tr>
<tr>
<td>unary compare</td>
<td><code>op x ?</code></td>
</tr>
<tr>
<td>binary compare</td>
<td><code>x op y ?</code></td>
</tr>
</tbody>
</table>

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

Control flow graph containing RTL instructions:

```
  i := 1

  if i <= 10 then
    t1 := i * 4
    t2 := i * b
    t3 := *(t2 + t1)
    t4 := t3 * 5
    t5 := i * 4
    t6 := &a
    *(t6 + t5) := t4
    i := i + 1
  end
```

Comparison

Advantages of high-level rep:
- analysis can exploit high-level knowledge of constructs
- probably faster to analyze
- easy to map to source code terms for debugging, profiling
- (may be) more compact

Advantages of low-level rep:
- can do low-level, machine-specific optimizations
  (if target-based representation)
- high-level rep may not be able to express some transformations
- can have relatively few kinds of instructions to analyze
- can be language-independent

High-level rep suitable for a source-to-source or special-purpose optimizer, e.g. inliner, parallelizer

Can mix multiple representations in single compiler
Can sequence compilers using different reps

Q: what about Java bytecodes?

Components of representation

Operations

Dependences between operations
- control dependences: sequencing of operations
  - evaluation of then & else arms depends on result of test
  - side-effects of statements occur in right order
- data dependences: flow of values from definitions to uses
  - operands computed before operation

Ideal: represent just those dependences that matter
- dependences constrain transformations
- fewest dependences => most flexibility in implementation
Representing control dependences

Option 1: **high-level representation**
- control flow implicit in semantics of AST nodes

Option 2: **control flow graph** (CFG)
- nodes are **basic blocks**
  - instructions in basic block sequence side-effects
- edges represent branches
  (control flow between basic blocks)

Option 2b: CFG whose nodes are individual instructions

Some fancier options:
- **control dependence graph**, part of **program dependence graph** (PDG) [Ferrante et al. 87]
- convert into data dependences on a memory state, in **value dependence graph** (VDG) [Weise et al. 94]

Representing data dependences

Option 1: **implicitly through variable defs/uses in CFG**
- simple, source-like
  - may overconstrain order of operations
  - analysis wants important things explicit \( \Rightarrow \) analysis can be slow

Option 2: **def/use chains**, linking each def to each use
- a kind of **data flow graph** (DFG)
  - explicit \( \Rightarrow \) analysis can be fast
  - must be computed, maintained after transformations
  - may be space-consuming

Some fancier options:
- **static single assignment** (SSA) form [Alpern et al. 88]
- value dependence graphs (VDGs)
  - ...

Example

```
1 \times := \ldots
2 \ldots \times \ldots
3 \ldots \times \ldots
4 \times := \ldots
5 \ldots \times \ldots
6 \ldots \times \ldots
7 \ldots \times \ldots
8 \ldots \times \ldots
```

```c
1
2
3
4
5
6
7
8
```