Northeastern University

CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- Big thank you to UW CSE faculty member, Hal Perkins
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Agenda

- Review – mapping source code to x86
  - Basic statements and expressions
  - Object representation and layout
  - Field access
  - What is this?
- More code shape
  - Object creation – new
  - Method calls
    - Dynamic dispatch
    - Method tables
    - super
  - Runtime type information
- Optimization and transformation
  - Survey of some code “optimizations” (improvements)
  - Some organizing concepts
    - Basic blocks
    - Control-flow and dataflow graph
    - Analysis vs. transformation
    - A closer look at some common optimizing transformations

Reading:

Cooper and Torczon, chapters 4.1-4.4, 5.5, 6.2-6.5 and 7.1-7.4, 8.1-8.6
Dragon book, chapters 6.3-6.5, 7.1-7.7, 8.1-8.4, 9.1

Code Shape Review

Review: Variables

- For us, all data is either:
  - In a stack frame (method local variables)
  - In an object (instance variables)
- Local variables accessed via %rbp

```
movq  -16(%rbp),%rax
```

- Object instance variables accessed via an offset from an object address in a register

  (details later today)

Review: Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos

- In the following, j_{false} is used to mean jump when a condition is false

  - No such instruction on x86-64
  - Will have to realize it with appropriate instruction to set condition codes followed by conditional jump
  - Normally don't need to actually generate the value "true" or "false" in a register
    - But this is a useful shortcut hack for the project

Review: While

- Source

  while (cond) stmt

- x86-64

```
test:  <code evaluating cond>
       jfalse done
       <code for stmt>
       jmp  test
done:
```

  – Note: In generated asm code we need to have unique labels for each loop, conditional statement, etc.

Review: Do-While

- Source

  do stmt while(cond)

- x86-64

```
loop:  <code for stmt>
       <code evaluating cond>
       j_true  loop
```

Review: If

- Source

    if (cond) stmt

- x86-64

```
        <code evaluating cond>
        j_false skip
        <code for stmt>
   skip:
```

Review: If-Else

- Source

  if (cond) $stmt_1$ else $stmt_2$

- x86-64

```
        <code evaluating cond>
        j_false else
        <code for stmt_1>
        jmp  done
 else: <code for stmt_2>
 done:
```

Review: Boolean Expressions

- What do we do with this?

    x > y

- Expression that evaluates to true or false
    - Could generate the value (0/1 or whatever the local convention is)
    - But normally we don't want/need the value – we're only trying to decide whether to jump

Review: Code for exp1 > exp2

- Basic idea: generated code depends on context:
  - What is the jump target?
  - Jump if the condition is true or if false?
- Example: evaluate exp1 > exp2, jump on false, target if jump taken is L123

```
<evaluate exp1 to %rax>
<evaluate exp2 to %rdx>
cmpq  %rdx,%rax
jng     L123
```

Review: Realizing Boolean Values

- If a boolean value needs to be stored in a variable or method call parameter, generate code needed to actually produce it
- Typical representations: 0 for false, +1 or -1 for true
  - C specifies 0 and 1 if stored; we'll use that
  - Best choice can depend on machine instructions; normally some convention is established during the primeval history of the architecture

0-Origin 1-D Integer Arrays Review

- Source

  $exp_1[exp_2]$

- x86-64

```
<evaluate exp1 (array address) in %rax>
<evaluate exp2 in %rdx>
address is (%rax,%rdx,8)  # if 8 byte
elements
```

2-D Arrays Review

- Subscripts start with 0
- C/C++, etc. specify row-major order
  - E.g., an array with 3 rows and 2 columns is stored in sequence: a(0,0), a(0,1), a(1,0), a(1,1), a(2,0), a(2,1)
- Fortran specifies column-major order
  - Exercises: What is the layout? How do you calculate location of a[i][j]? What happens when you pass array references between Fortran and C/C++ code?
- Java does not have “real” 2-D arrays. A Java 2-D array is a pointer to a list of pointers to the rows
  - And rows may have different lengths (ragged arrays)

More Code Shape

What does this program print?

```
class One {
    int tag;
    int it;

    void setTag()   {
    tag = 1; }

    int getTag() {
    return tag; }

    void setIt(int
    it) {
    this.it = it; }

    int getIt() {
    return it; }
}
```

```
class Two extends
    One {
    int it;
    void setTag() {
     tag = 2;
     it = 3;
    }

    int getThat() {
     return it;
    }

    void resetIt() {
     super.setit(42);
    }
}
```

```
public static void main(String[] args)
    {
        Two two = new Two();
        One one = two;

        one.setTag();

    System.out.println(one.getTag());

        one.setIt(17);
        two.setTag();

    System.out.println(two.getIt());

    System.out.println(two.getThat());
        two.resetIt();

    System.out.println(two.getIt());

    System.out.println(two.getThat());

    }
```

Object Representation

- The naïve explanation is that an object contains
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides (shadows) superclass instance – but the superclass field is still there
  - Methods declared in its class and all superclasses
    - Redeclaration of a method overrides (replaces) – but overridden methods can still be accessed by super…
- When a method is called, the method “inside” that particular object is called
  - Regardless of the static (compile-time) type of the variable
    - (But we really don’t want to copy all those methods, do we?)

Actual Representation

- Each object contains:
  - An entry ("slot") for each field (instance variable)
    - Including shadowed and private fields in superclasses
  - A pointer to a runtime data structure for its class
    - Key component: method dispatch table (next slide)
- Basically a C struct
- Fields hidden by declarations in subclasses are *still* allocated in the object and are accessible from superclass methods

Method Tables and Inheritance

- A really simple implementation
  - Method table for each class has pointers to all methods declared in it (a dictionary)
  - Method table also contains a pointer to parent class method table
  - Method dispatch:
    - Look in current table and use if method declared locally
    - Look in parent class table if not local
    - Repeat
    - "Message not understood" if you can't find it after search
  - Actually used/needed in typical implementations of some dynamic languages (e.g. Ruby, Smalltalk, etc.)

O(1) Method Dispatch

- Idea: first part of method table for extended class has pointers for the same methods in the same order as the parent class
  – BUT pointers actually refer to overriding methods if these exist

  ∴ Method dispatch can be done with indirect jump using fixed offsets known at compile time – O(1)
    - In C: `*(object->vtbl[offset])(parameters)`
- Pointers to methods added in subclass are after ptrs to inherited/overridden ones in vtable

Now What?

- Need to explore
  - Object layout in memory
  - Compiling field references
    - Implicit and explicit use of “this”
  - Representation of vtables
  - Object creation – new
  - Code for dynamic dispatch
  - Runtime type information – instanceof and casts

Object Layout

- Typically, allocate fields sequentially
- Follow processor/OS alignment conventions for struct/object when appropriate/available
  - Include padding bytes for alignment as needed
- Use first word of object for pointer to method table/class information
- Objects are allocated on the heap
  - No actual bits in the generated code

Object Field Access

- Source

  int n = obj.slot;

- x86-64
  - Assuming that obj is a local variable in the current method's stack frame

    ```
    movq    offset_obj(%rbp),%rax       # load obj ptr
    movq    offset_slot(%rax),%rax      # load slot
    movq    %rax,offset_n(%rbp)         # store n
    ```

  - Same idea used to reference fields of "this"
    - Use implicit "this" parameter passed to method instead of a local variable to get object address

Local Fields

- A method can refer to fields in the receiving object either explicitly as “this.f” or implicitly as “f”
  - Both compile to the same code – an implicit “this.” is assumed if not present explicitly

Source Level View

What you write:

```
int getIt() {
  return it;
}
void setIt(int it) {
  this.it = it;
}
…
obj.setIt(42);
k = obj.getIt();
```

What you really get:

```
int getIt(ObjType this) {
  return this.it;
}
void setIt(ObjType this, int it) {
  this.it = it;
}
…
setIt(obj, 42);
k = getIt(obj);
```

x86-64 “`this`” Convention (C++)

- “`this`” is an implicit first parameter to every non-static method
- Address of object placed in %rdi for every non-static method call
- Remaining parameters (if any) in %rsi, etc.

- We’ll use this convention in our project

MiniJava Method Tables (vtbls)

- Generate these as initialized data in the assembly language source program
- Need to pick a naming convention for assembly language labels; suggest:
  - For methods, classname$methodname
    - Would need something more sophisticated for overloading
  - For the vtables themselves, classname$$
- First method table entry points to superclass table (we might not use this in our project, but is helpful if you add instanceof or type cast checks)

Method Tables For Convoluted Example (gcc/as syntax)

```
class One {
   void setTag()     { … }
   int getTag()       { … }
   void setIt(int it) {…}
   int getIt()          { … }
}

class Two extends One {
   void setTag() { … }
   int getThat() { … }
   void resetIt() { … }
}
```

```
                 .data

One$$: .quad   0          # no superclass
       .quad   One$setTag
       .quad   One$getTag
       .quad   One$setIt
       .quad   One$getIt

Two$$: .quad   One$$      # superclass
       .quad   Two$setTag
       .quad   One$getTag
       .quad   One$setIt
       .quad   One$getIt
       .quad   Two$getThat
       .quad   Two$resetIt
```

Method Table Layout

Key point: first method entries in `Two's` method table are pointers to methods declared in `One` in *exactly the same order*

- Actual pointers reference code appropriate for objects of each class (inherited or overridden)

∴ Compiler knows correct offset for a particular method pointer *regardless of whether that method is overridden* and regardless of the actual (dynamic) type of the object

Object Creation – `new`

Steps needed

- Call storage manager (malloc or similar) to get the raw bits
  - (and store 0's if required by the language, e.g., Java)
- Store pointer to method table in the first 8 bytes of the object
- Call a constructor with "`this`" pointer to the `new` object in %rdi and other parameters as needed
  - (Not in MiniJava since we don't have constructors)
- Result of `new` is a pointer to the `new` object

Object Creation

- Source

```
One one = new One(…);
```

- x86-64

```
movq    $nBytesNeeded,%rdi       # obj size + 8 (include space for
vtbl ptr)
call        mallocEquiv              # addr of allocated bits
returned in %rax
leaq        One$$,%rdx               # get method table address
movq    %rdx,0(%rax)                 # store vtbl ptr at beginning
of object
movq    %rax,%rdi                # set up "this" for constructor
movq    %rax,offsettemp(%rbp)    # save "this" for later
<load constructor arguments>         # arguments (if needed)
call        One$One                  # call ctr if we have one (no
vtbl lookup)
movq    offsettemp(%rbp),%rax        # recover ptr to object
movq    %rax,offsetone(%rbp)     # store object reference in
variable
```

Constructor

- Why don’t we need a vtable lookup to find the right constructor to call?
- Because at compile time we know the actual class (it says so right after “new”), so we can generate a call instruction to a known label
  - Same with super.method(…) or superclass constructor calls – at compile time we know all of the superclasses (need this to compile subclass and construct method tables), so we know statically what class “super.method” belongs to

Method Calls

- Steps needed
  - Parameter passing: just like an ordinary C function, except load a pointer to the object in %rdi as the first ("this") argument
  - Get a pointer to the object's method table from the first 8 bytes of the object
  - Jump indirectly through the method table

Method Call

- Source

  obj.m(…);

- x86-64

```
<load arguments in registers as usual>  # as needed
movq    offset_obj(%rbp),%rdi # first argument is obj ptr ("this")
movq    0(%rdi),%rax          # load vtable address into %rax
call    *offset_m(%rax)       # call function whose address is at
                              #      known offset in the vtable *
```

- * Or can use: addq $offset_m,%rax
  call *(%rax)

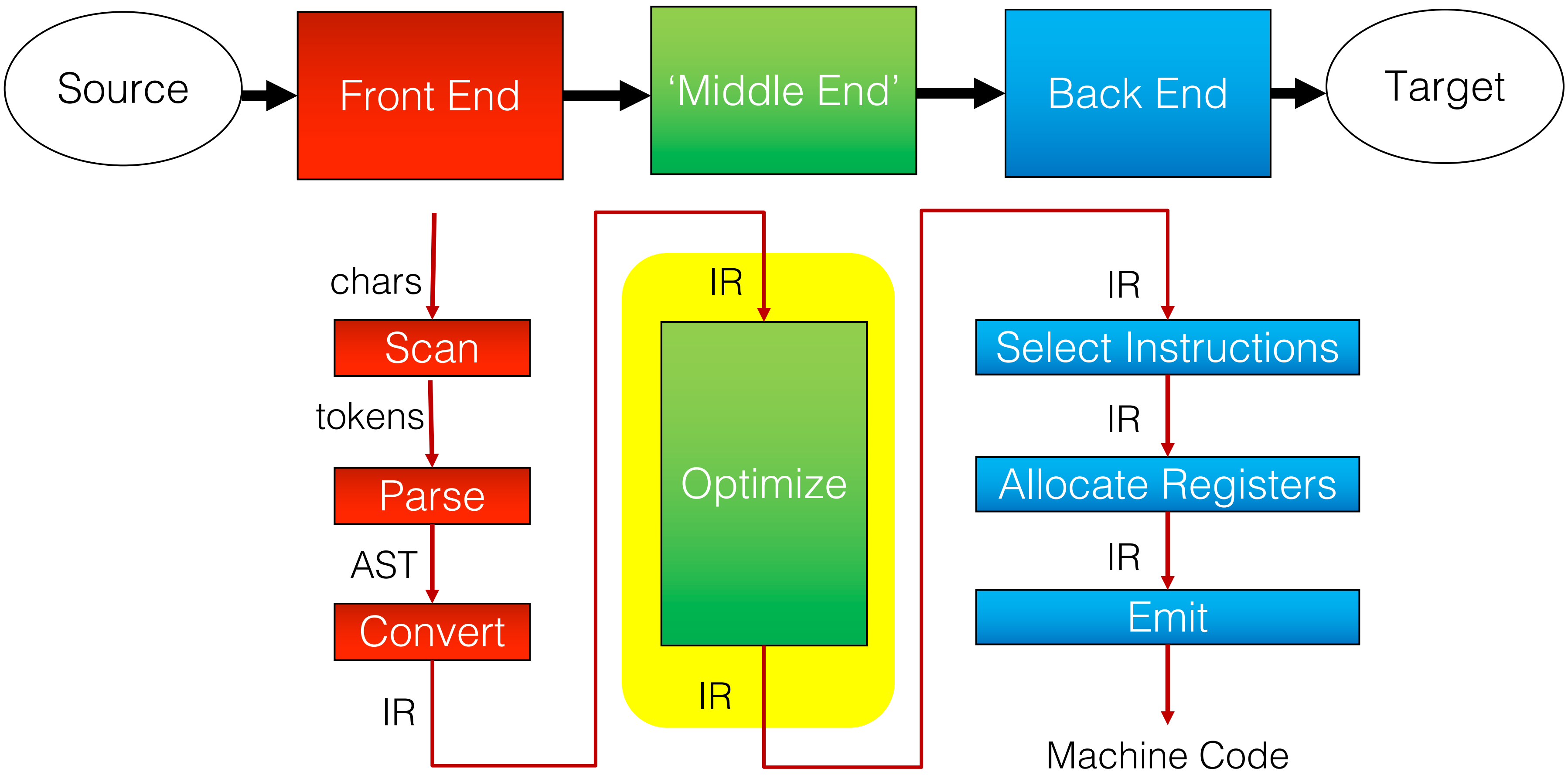
  or : movq $offset_m(%rax),%rax
  call *%rax

Runtime Type Checking

- We can use the method table for the class as a "runtime representation" of the class
  - Each class has one vtable at a unique address
- The test for "o instanceof C" is
  - Is o's method table pointer == &C$$?
    - If so, result is "true"
  - Recursively, get pointer to superclass method table from the method table and check that
  - Stop when you reach Object (or a null pointer, depending on whether there is a ultimate superclass of everything)
    - If no match by the top of the chain, result is "false"
- Same test as part of check for legal downcast (e.g., how to test for ClassCastException in (type)obj cast)

Optimization and Transformation

Optimizations in a Compiler



AST = Abstract Syntax Tree

IR = Intermediate Representation

Optimizations

- Use added passes to identify inefficiencies in intermediate or target code
- Replace with equivalent but better sequences
  - Equivalent = “has same externally visible behavior”
  - Better can mean many things: faster, smaller, less power, etc.
- “Optimize” overly optimistic: “usually improve” is generally more accurate
  - And “clever” programmers can outwit you!

Kinds of Optimizations

- Peephole - look at adjacent instructions
- Local - look at individual *basic blocks*
  - straight-line sequence of statements
- Intraprocedural - look at the whole procedure
  - Commonly called "global"
- Interprocedural - look across procedures
  - "whole program" analysis
  - gcc's "link time optimization" is a version of this
- Larger scope - usually better optimization but more cost and complexity
  - Analysis is often less precise because of more possibilities

Peephole Optimization

- Look at adjacent instructions (a “peephole” on the code stream)
  - try to replace adjacent instructions with something faster

`movq %r9,16(%rsp)` `movq 16(%rsp),%r12`	`movq %r9,16(%rsp)` `movq %r9,%r12`

  - Jump chaining can also be considered a form of peephole optimization (removing jump to jump)

More Examples

`subq $8,%rax`
`movq %r2,0(%rax)`
`# %rax overwritten`
	`movq %r2,-8(%rax)`

`movq 16(%rsp),%rax`
`addq $1,%rax`
`movq %rax,16(%rsp)`
`# %rax overwritten`
	`incq 16(%rsp)`

- One way to do complex instruction selection

Algebraic Simplification

- "constant folding", "strength reduction"
  - z = 3 + 4; ➔ z = 7
  - z = x + 0; ➔ z = x
  - z = x * 1; ➔ z = x
  - z = x * 2; ➔ z = x << 1 or z = x + x
  - z = x * 8; ➔ z = x << 3
  - z = x / 8; ➔ z = x >> 3 (only if x>=0 known)
  - z = (x + y) - y; ➔ z = x (maybe; not doubles, might change int overflow)
- Can be done at many levels from peephole on up
- Why do these examples happen?
  - Often created during conversion to lower-level IR, by other optimizations, code gen, etc.

Local Optimizations

- Analysis and optimizations within a basic block
- *Basic block*: straight-line sequence of statements
  - no control flow into or out of middle of sequence
- Better than peephole
- Not too hard to implement with reasonable IR

- Machine-independent, if done on IR

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; unoptimized intermediate code:

`count = 10;`
`...   // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = count;`
	`t2 = 5;`
	`t3 = t1 * t2;`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4,t5);`
	`y = t6;`
	`x = 7`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; constant propagation:

`count = 10;`
`...  // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = 10;        // cp count`
	`t2 = 5;`
	`t3 = 10 * t2;   // cp t1`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4,3);  // cp t5`
	`y = t6;`
	`x = 7`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; constant folding:

`count = 10;`
`...   // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;         // 10*t2`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4,3);`
	`y = t6;`
	`x = 7;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; repropagated intermediate code

`count = 10;`
`...   // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;      // cp t3`
	`t4 = 50;     // cp x`
	`t5 = 3;`
	`t6 = exp(50,3); // cp t4`
	`y = t6;`
	`x = 7;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; refold intermediate code

`count = 10;`
`...  // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000; // cf 50^3`
	`y = t6;`
	`x = 7;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable reassigned)
- Can enable more constant folding
  - Code; repropagated intermediate code

`count = 10;`
`...  // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;`
	`y = 125000;  // cp t6`
	`x = 7;`

Local Dead Assignment Elimination

- If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
    Clean-up after previous optimizations, often

`count = 10;`
`...  // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;`
	`y = 125000;`
	`x = 7;`

Local Dead Assignment Elimination

- If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
    Clean-up after previous optimizations, often

`count = 10;`
`... // count not changed`
`x = count * 5;`
`y = x ^ 3;`
`x = 7;`
	`count = 10;`
	~~`t1 = 10;`~~
	~~`t2 = 5;`~~
	~~`t3 = 50;`~~
	~~`x = 50;`~~
	~~`t4 = 50;`~~
	~~`t5 = 3;`~~
	~~`t6 = 125000;`~~
	`y = 125000;`
	`x = 7;`

Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won't have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

```
... a[i] + b[i] ...
```

```
t1 = *(fp + ioffset);
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);
t5 = *(fp + ioffset);
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);
t9 = t4 + t8;
```

Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won't have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;   // CSE`
	`t6 = t5 * 4;`
	`t7 = fp + t6;`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won't have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;`
	`t6 = t1 * 4;  // CP`
	`t7 = fp + t6;`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won't have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;`
	`t6 = t2;      // CSE`
	`t7 = fp + t2; // CP`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won't have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;`
	`t6 = t2;`
	`t7 = t3;  // CSE`
	`t8 = *(t3 + boffset); //CP`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Look for repetitions of the same computation. Eliminate them if result won't have changed and no side effects
  - Avoid repeated calculation and eliminates redundant loads
- Idea: walk through basic block keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	~~`t5 = t1;`~~ `// DAE`
	~~`t6 = t2;`~~ `// DAE`
	~~`t7 = t3;`~~ `// DAE`
	`t8 = *(t3 + boffset);`
	`t9 = t4 + t8;`

Intraprocedural Optimizations

- Enlarge scope of analysis to whole procedure
  - more opportunities for optimization
  - have to deal with branches, merges, and loops
- Can do constant propagation, common subexpression elimination, etc. at “global” level
- Can do new things, e.g. loop optimizations
- Optimizing compilers usually work at this level (-O2)

Code Motion

- Goal: move loop-invariant calculations out of loops
- Can do at source level or at intermediate code level

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + b[j];
  z = z + 10000;
}
```

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + t1;
  z = z + t2;
}
```

Code Motion at Intermediate Level

```
for (i = 0; i < 10; i = i+1) {
  a[i] = b[j];
}
```

```
 *(fp + ioffset) = 0;
label top;
  t0 = *(fp + ioffset);
  iffalse (t0 < 10) goto done;
  t1 = *(fp + joffset);
  t2 = t1 * 4;
  t3 = fp + t2;
  t4 = *(t3 + boffset);
  t5 = *(fp + ioffset);
  t6 = t5 * 4;
  t7 = fp + t6;
  *(t7 + aoffset) = t4;
  t9 = *(fp + ioffset);
  t10 = t9 + 1;
  *(fp + ioffset) = t10;
  goto top;
label done;
```

Code Motion at Intermediate Level

```
for (i = 0; i < 10; i = i+1) {
  a[i] = b[j];
}
```

```
t11 = fp + ioffset; t13 = fp + aoffset;
t12 = fp + joffset; t14 = fp + boffset
*(fp + ioffset) = 0;
label top;
  t0 = *t11;
  iffalse (t0 < 10) goto done;
  t1 = *t12;
  t2 = t1 * 4;
  t3 = t14;
  t4 = *(t14 + t2);
  t5 = *t11;
  t6 = t5 * 4;
  t7 = t13;
  *(t13 + t6) = t4;
  t9 = *t11;
  t10 = t9 + 1;
  *t11 = t10;
  goto top;
label done;
```

Loop Induction Variable Elimination

- A special and common case of loop-based strength reduction
- For-loop index is *induction variable*
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, can rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop
  - can then do loop-invariant code motion

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + x;
}
=> transformed to
for (p = &a[0]; p < &a[10]; p = p+4) {
  *p = *p + x;
}
```

Interprocedural Optimization

- Expand scope of analysis to procedures calling each other
- Can do local & intraprocedural optimizations at larger scope
- Can do new optimizations, e.g. inlining

Inlining: Replace Call With Body

- Replace procedure call with body of called procedure
- Source:

```
final double pi = 3.1415927;
double circle_area(double radius) {
  return pi * (radius * radius);
}
...
double r = 5.0;
...
double a = circle_area(r);
```

- After inlining:

```
...
double r = 5.0;
...
double a = pi * r * r;
```

- (Then what? Constant propagation/folding)

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 = *(fp + ioffset);  // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset); // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Strength reduction: shift often cheaper than multiply

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;  // was t1 * 4
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = t5 << 2;  // was t5 * 4
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset); // i
t14 = t13 << 2; // was t13 * 4
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant propagation: replace variables with known constant values

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = 2 << 2;  // was t5 << 2
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - 5;  // was t10 – t11
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Dead store (or dead assignment) elimination: remove assignments to provably unused variables

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = 2 << 2;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 – 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant folding: statically compute operations with known constant values

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 8;  // was 2 << 2
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant propagation then dead store elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
```
~~t6 = 8;~~
```
t7 = fp + 8;  // was fp + t6
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Arithmetic identities: + is commutative & associative. boffset is typically a known, compile-time constant (say -32), so this enables…

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = boffset + 8;  // was fp + 8
t8 = *(t7 + fp);  // b[2] (was t7 + boffset)
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

… more constant folding, which in turn enables …

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = -24;          // was boffset (-32) + 8
t8 = *(t7 + fp);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

More constant propagation and dead store elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
```
~~t7 = -24;~~
```
t8 = *(fp - 24);  // b[2]  (was t7+fp)
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Common subexpression elimination – no need to compute *(fp+ioffset) again if we know it won't change

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = *(fp + xoffset); // x
t12 = t10 – 5;
t13 = t1;        // i  (was *(fp + ioffset))
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Copy propagation: replace assignment targets with their values (e.g., replace t13 with t1)

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = t9;       // x (was *(fp + xoffset))
t12 = t10 – 5;
t13 = t1;              // i
t14 = t1 << 2;  // was t13 << 2
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Common subexpression elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = t9;              // x
t12 = t10 – 5;
t13 = t1;              // i
t14 = t2;        // was t1 << 2
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

More copy propagation

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = t9;              // x
t12 = t9 – 5;    // was t10 - 5
t13 = t1;              // i
t14 = t2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

More copy propagation

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t10 = t9;              // x
t12 = t9 – 5;
t13 = t1;              // i
t14 = t2;
t15 = fp + t2;  // was fp + t14
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Dead assignment elimination

t1 = *(fp + ioffset); // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset); // a[i]
t8 = *(fp - 24); // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
~~t10 = t9; // x~~
t12 = t9 – 5;
~~t13 = t1; // i~~
~~t14 = t2;~~
t15 = fp + t2;
*(t15 + coffset) = t12; // c[i] := …

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t12 = t9 – 5;
t15 = fp + t2;
*(t15 + coffset) = t12; // c[i] := …
```

- Final: 3 loads (i, a[i], b[2]), 2 stores (x, c[i]), 5 register-only moves, 9 +/-, 1 shift
- Original: 5 loads, 2 stores, 10 register-only moves, 12 +/-, 3 *

- Optimizer note: we usually leave assignment of actual registers to later stage of the compiler and assume as many “pseudo registers” as we need here

Some Frequent Compiler Optimization Techniques

- Strength reduction – replace an “expensive” operation with an equivalent, but less expensive operation (e.g., multiplication → summation/shift)
- Constant propagation – substitute values of known constants at compile time
- Constant folding – recognize and evaluate a constant at compile time rather than run tie
- Dead assignment elimination – recognize assignments that never referenced, and remove them from the code
- Common subexpression elimination - find repetitions of same computations, and eliminate them if result won’t changed
- **Code motion** - move loop-invariant calculations out of loops
- Inlining – replace some function calls with the body of the function (e.g., some getters)

Data Structures for Optimizations

- Need to represent control and data flow
- Control flow graph (CFG) captures flow of control:
  - nodes are IL statements, or whole basic blocks
  - edges represent (all possible) control flow
  - node with multiple successors = branch/switch
  - node with multiple predecessors = merge
  - loop in graph = loop
- Data flow graph (DFG) captures flow of data (e.g. def/use chains):
  - nodes are def(inition)s and uses
  - edge from def to use
  - a def can reach multiple uses
  - a use can have multiple reaching defs (different control flow paths, possible aliasing, etc.)
- SSA: another widely used way of linking defs and uses

Summary

- Optimizations organized as collections of passes, each rewriting IL in place into (hopefully) better version
- Each pass does analysis to determine what is possible, followed by transformation(s) that (hopefully) improve the program
  - Sometimes “analysis-only” passes are helpful
  - Often redo analysis/transformations again to take advantage of possibilities revealed by previous changes
- Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write

Analysis and Transformation

Analysis and Transformation

- Each optimization is made up of
  - Some number of analyses
  - Followed by a transformation
- Analyze CFG and/or DFG by propagating info forward or backward along CFG and/or DFG edges
  - Merges in graph require combining info
  - Loops in graph require *iterative approximation*
- Perform (improving) transformations based on info computed
- Analysis must be conservative/safe/sound so that transformations preserve program behavior

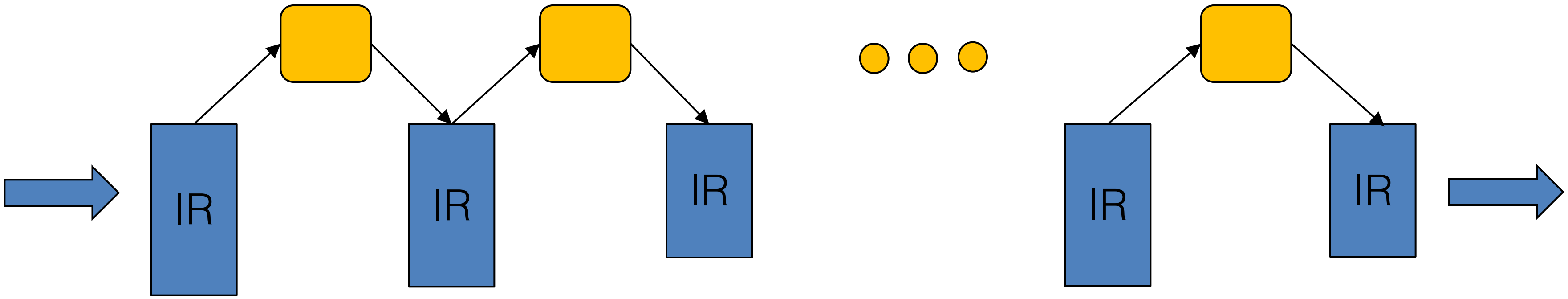
Role of Transformations

- Dataflow analysis discovers opportunities for code improvement
- Compiler rewrites the (IR) to make these improvements
  - Transformation may reveal additional opportunities for further optimization
  - May also block opportunities by obscuring information

Organizing Transformations in a Compiler

- Typically middle end consists of many phases
  - Analyze IR
  - Identify optimization
  - Rewrite IR to apply optimization
  - And repeat (50 phases in a commercial compiler is typical)
- Each individual optimization is supported by rigorous formal theory
- But no formal theory for what order or how often to apply them(!)
  - Some rules of thumb and best practices
  - May apply some transformations several times as different phases reveal opportunities for further improvement

Optimization 'Phases'



- Each optimization requires a 'pass' (linear scan) over the IR
- IR may sometimes shrink, sometimes expand
- Some optimizations may be repeated
- 'Best' ordering is heuristic
- Don't try to *beat* an optimizing compiler - you will lose!

- Note: not all programs are written by humans!
- Machine-generated code can pose a challenge for optimizers
  - eg: a single function with 10,000 statements, 1,000+ local variables, loops nested 15 deep, spaghetti of "GOTOs", etc

A Taxonomy

- Machine Independent Transformations
  - Mostly independent of target machine
    (e.g., loop unrolling will likely make it faster regardless of target)
  - "Mostly"? – e.g., vectorize only if target has SIMD ops
  - Worthwhile investment – applies to all targets
- Machine Dependent Transformations
  - Mostly concerned with instruction selection & scheduling, register allocation
  - Need to tune for different targets
  - Most of this in the back end, but some in the optimizer

Machine Independent Transformations

- Dead code elimination
  - unreachable or not actually used later
- Code motion
  - “hoist” loop-invariant code out of aloop
- Specialization
- Strength reduction
  - 2*x => x+x; @A+((i*numcols+j)*eltsize => p+=4
- Enable *other* transformations
- Eliminate redundant computations
  - Value numbering, GCSE

Machine Dependent Transformations

- Take advantage of special hardware
  - e.g., expose instruction-level parallelism (ILP)
  - e.g., use special instructions (VAX polyf; x86 sqrt, strings)
  - e.g., use SIMD instructions and registers
- Manage or hide latencies
  - e.g., tiling/blocking and loop interchange
  - Improves cache behavior – hugely important
- Deal with finite resources - # functional units
- Compilers generate for a vanilla machine, e.g., SSE2
  - But provide switches to tune (arch:AVX, arch:IA32)
  - JIT compiler knows its target architecture!

Optimizer Contracts

- Prime directive
  - No optimization will change observable program behavior!
  - This can be subtle. e.g.:
    - What is "observable"? (via IO? to another thread?)
    - Dead-Code-Eliminate a *throw*?
    - Language Reference Manual may be ambiguous/undefined/negotiable for edge cases
- Avoid harmful optimizations
  - If an optimization does not improve code significantly, don't do it: it harms throughput
  - If an optimization degrades code quality, don't do it

Is this *hoist* legal?

```
for (int i = start; i < finish; ++i) a[i] += 7;
```

```
    i = start
loop:
    if (i >= finish) goto done
    if (i < 0 || i >= a.length) throw OutOfBounds
    a[i] += 7
    goto loop
done:
```

```
    if (start < 0 || finish >= a.length) throw OutOfBounds
    i = start
loop:
    if (i >= finish) goto done
    a[i] += 7
    goto loop
done:
```

Another example: "volatile" pretty much kills all attempts to optimize

Dead Code Elimination

- If a compiler can prove that a computation has no external effect, it can be removed
  - Unreachable operations – always safe to remove
  - Useless operations – reachable, may be executed, but results not actually required
- Dead code often results from other transformations
  - Often want to do DCE several times

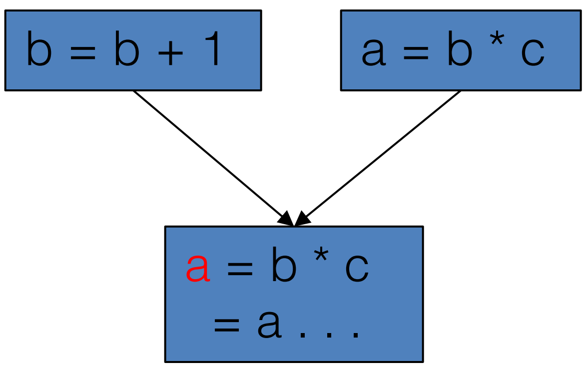
Dead Code Elimination

- Classic algorithm is similar to garbage collection
  - Pass I – Mark all useful operations
    - Instructions whose result does, or can, affect visible behavior:
      - Input or Output
      - Updates to object fields that might be used later
      - Instructions that may throw an exception (e.g.: array bounds check)
      - Calls to functions that might perform IO or affect visible behavior
      - (Remember, for many languages, compiler does not process entire program at one time – but a JIT compiler might be able to)
    - Mark all useful instructions
    - Repeat until no more changes
  - Pass II – delete all unmarked operations

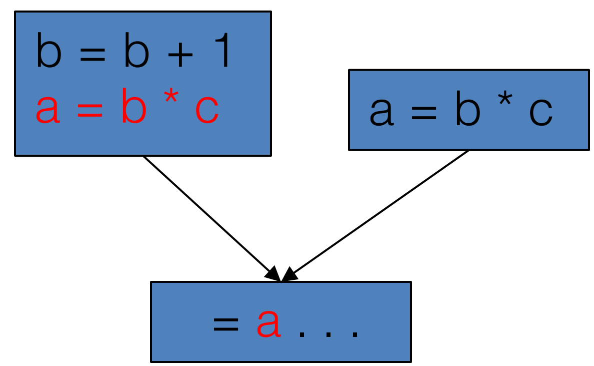
Code Motion

- Idea: move an operation to a location where it is executed less frequently
  - Classic situation: *hoist* loop-invariant code: execute once, rather than on every iteration

- Lazy code motion & *partial* redundancy



a must be re-calculated - wasteful if control took right-hand arm



Replicate, so a need not be re-calculated

Specialization I

- Idea: replace general operation in IR with more specific
  - Constant folding:
    - feet_per_minute = mph * feet_per_mile/minutes_per_hour
    - feet_per_minute = mph * 5280 / 60
    - feet_per_minute = mph * 88
  - Replacing multiplications and division by constants with shifts (when safe)
  - Peephole optimizations
    - movl $0,%eax => xorl %eax,%eax

Specialization:2 - Eliminate Tail Recursion

- Factorial - recursive
  int fac(n) = if (n <= 2) return 1; else return n * fac(n - 1);
- 'accumulating' Factorial - tail-recursive
  facaux(n, r) = if (n <= 2) return 1; else return facaux(n - 1, n*r)
  call facaux(n, 1)
- Optimize-away the call overhead; replace with simple jump
  facaux(n, r) = if (n <= 2) return 1;
  else n = n - 1; r = n*r; jump back to start of facaux
  - So replace recursive call with a loop and just one stack frame
- Issue?
  - Avoid stack overflow - good! - "observable" change?

Strength Reduction

- Classic example: Array references in a loop

  `for (k = 0; k < n; k++) a[k] = 0;`

- Naive codegen for a[k] = 0 in loop body

```
movl $4,%eax                  // elemsize = 4 bytes
imull offset_k(%rbp),%eax     // k * elemsize
addl  offset_a(%rbp),%eax     // &a[0] + k * elemsize
mov  $0,(%eax)                // a[k] = 0
```

- Better!

```
movl offset_a(%rbp),eax       // &a[0], once-off

movl $0,(%eax)                // a[k] = 0
addl $4,%eax                  // eax = &a[k+1]
```

Note: *pointers* allow a user to do this directly in C or C++
Eg: for (p = a; p < a + n;) *p++ = 0;

Implementing Strength Reduction

- Idea: look for operations in a loop involving:
  - A value that does not change in the loop, the *region constant*, and
  - A value that varies systematically from iteration to iteration, the *induction variable*
- Create a new induction variable that directly computes the sequence of values produced by the original one; use an addition in each iteration to update the value

Other Common Transformations

- Inline substitution (procedure bodies)
- Cloning / Replicating
- Loop Unrolling
- Loop Unswitching

Inline Substitution - "inlining"

Class with trivial *getter*

```
class C {
  int x;
  int getx() { return x; }
}
```

Method f calls getx

```
class X {
  void f() {
    C c = new C();
    int total = c.getx() + 42;
  }
}
```

Compiler *inlines* body of getx into f

```
class X {
  void f() {
    C c = new C();
    int total = c.x + 42;
  }
}
```

- Eliminates call overhead
- Opens opportunities for more optimizations
- Can be applied to large method bodies too
- Aggressive optimizer will inline 2 or more deep
- Increases total code size (memory & cache issues)
- With care, is a huge win for OO code

Code Replication

Original

```
if (x < y) {
   p = x + y;
} else {
   p = z + 1;
}
q = p * 3;
w = y + x;
```

Replicated code

```
if (x < y) {
   p = x + y;
   q = p * 3;
   w = y + x;
} else {
   p = z + 1;
   q = p * 3;
   w = y + x;
}
```

- + : extra opportunities to optimize in larger basic blocks (eg: LVN)
- - : increase total code size - may impact effectiveness of I-cache

Loop Unrolling

- Idea: replicate the loop body
  - More opportunity to optimize loop body
  - Increases chances for good schedules and instruction level parallelism
  - Reduces loop overhead (reduce test/jumps by 75%)
- Catches
  - must ensure unrolled code produces the same answer: "loop-carried dependency analysis"
  - code bloat
  - don't overwhelm registers

Loop Unroll Example

Original

```
for (i = 1, i <= n, i++) {
    a[i] = a[i] + b[i];
}
```

- Unroll 4x
- Need tidy-up loop for remainder

Unrolled

```
i = 1;
while (i + 3 <= n) {
  a[i]   = a[i]   + b[i];
  a[i+1] = a[i+1] + b[i+1];
  a[i+2] = a[i+2] + b[i+2];
  a[i+3] = a[i+3] + b[i+3];
  i += 4;
}

while (i <= n) {
    a[i] = a[i] + b[i];
    i++;
}
```

Loop Unswitching

- Idea: if the condition in an if-then-else is loop invariant, rewrite the loop by pulling the if-then-else out of the loop and generating a tailored copy of the loop for each half of the new conditional
  - After this transformation, both loops have simpler control flow – more chances for rest of compiler to do better

Loop Unswitch Example

Original

```
for (i = 1, i <= n, i++) {
     if (x > y) {
      a[i] = b[i]*x;
   } else {
      a[i] = b[i]*y;
   }
}
```

Unswitched

```
if (x > y) {
  for (i = 1; i <= n; i++) {
     a[i] = b[i]*x;
  }
} else {
  for (i = 1; i <= n; i++) {
     a[i] = b[i]*y;
  }
}
```

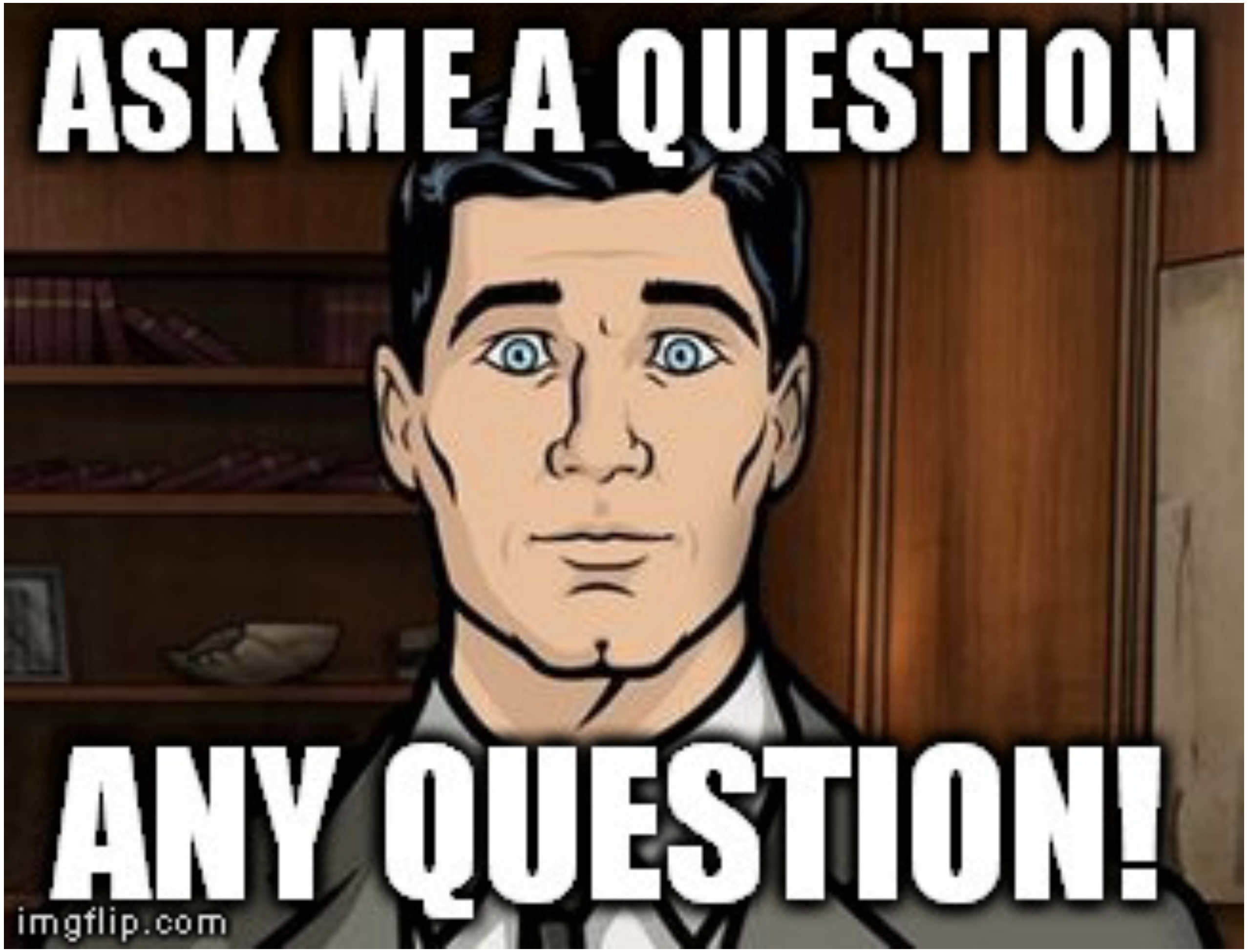
- IF condition does not change value in this code snippet
- No need to check x > y on every iteration
- Do the IF check once!

Summary

- Just a sampler
  - 100s of transformations in the literature
  - Will examine several in more detail, particularly involving loops
- Big part of engineering a compiler is:
  - decide which transformations to use
  - decide in what order
  - decide if & when to repeat each transformation
- Compilers offer options:
  - optimize for speed
  - optimize for codesize
  - optimize for specific target micro-architecture
  - optimize for power consumption(!)
- Competitive bench-marking will investigate many permutations

What's next

- Careful look at several analysis and transformation algorithms
- Value numbering / dominators
- Dataflow
- Loops, loops, loops
  - Dominators – discovering loop structures
  - Loop-invariant code
  - Loop Transformations



[Meme credit: imgflip.com]