Optimization

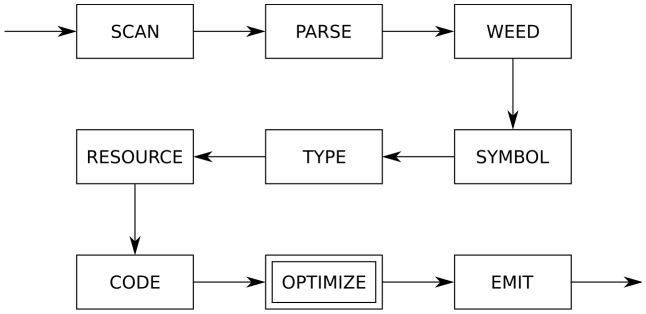
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 9:30-10:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2018/



Bob, from Accounting

Announcements (Friday, February 23rd)

Milestones

- Office hours tomorrow from 1:00-3:00 PM

Milestone 1

- Any questions?
- **Due:** Tuesday, February 27th 11:59 PM

Optimization

The *optimizer* focuses on

- Reducing the execution time (what you typically think of); or
- Reducing the code size; or
- Reducing the power consumption (new).

These goals often conflict, since a larger program may in fact be faster.

The best optimizations achieve all goals – but this is difficult to accomplish in general.

Optimizations for Space

Optimizations for *space* reduce the code size by replacing sequences of instructions with a smaller set

- Historically very important, because memory was small and expensive;
- When memory became large and cheap, optimizing compilers traded space for speed; but
- Then Internet bandwidth was small and expensive, so Java compilers optimized for space; but
- Today Internet bandwidth is larger and cheaper, so we optimize for speed again.

$\Rightarrow$ Optimizations are driven by economy!

Optimizations for Speed

Optimizations for *speed* improve the execution performance of the program. These types of optimizations form the bulk of modern optimizing compilers

- Historically very important to gain acceptance for high-level languages;
- Are still important, since the software always strains the limits of the hardware;
- Are challenged by ever higher abstractions in programming languages; and
- Must constantly adapt to changing microprocessor architectures.

Common optimization areas

- Cache performance;
- Parallel/vectorization;
- Loop invariants;
- Common-expressions removal (CSE)/dead code removal; and
- . . .

Optimization Passes

Optimizations may take place at various levels of program transformation/execution

- At the source code level;
- In an intermediate representation;
- At the binary machine code level; or
- At run-time (e.g. JIT compilers).

An aggressive optimization requires many small contributions from all levels.

Considerations

Choosing an optimization strategy is a balance between

- Compilation time;
- Execution time;
- Available information/representations (low-level/high-level);
- Runtime vs offline; and more

Optimization from a Programmer's Perspective

Should you program in "Optimized C"?

If you want a fast C program, should you use `LOOP #1` or `LOOP #2`?

```
/* LOOP #1 */
for (i = 0; i < N; i++) {
   a[i] = a[i] * 2000;
   a[i] = a[i] / 10000;
}

/* LOOP #2 */
b = a;
for (i = 0; i < N; i++) {
   *b = *b * 2000;
   *b = *b / 10000;
   b++;
}
```

What would the expert programmer do?

Optimization from a Programmer's Perspective

If you said `LOOP #2` ... you were wrong!

`LOOP`	opt. level	SPARC	MIPS	Alpha
`#1` (array)	no opt	20.5	21.6	7.85
`#1` (array)	opt	8.8	12.3	3.26
`#1` (array)	super	7.9	11.2	2.96
`#2` (ptr)	no opt	19.5	17.6	7.55
`#2` (ptr)	opt	12.4	15.4	4.09
`#2` (ptr)	super	10.7	12.9	3.94

- Pointers confuse most C compilers; don't use pointers instead of array references.
- Compilers do a good job of register allocation; don't try to allocate registers in your C program.
- In general, write clear C code; it is easier for both the programmer and the compiler to understand.

Optimization: Smaller and Faster

Intuitively, reducing the number of instructions needed to run a program can improve program performance

- Remove unnecessary operations;
- Simplify control structures; and
- Replace complex operations by simpler ones (strength reduction).

This is what the JOOS peephole optimizer does.

Later, we shall look at

- Parallelism through GPUs;
- JIT compilers (high level); and
- More powerful optimizations based on static analysis (COMP 621).

Optimization: Larger and Faster (Tabulation)

In some instances, expanding the code size can improve performance. Tabulation is one such approach which replaces function calls with an approximation

Sine function

$$\texttt{sin}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots...$$

Optimization using a lookup table

sin(0.0)	0.000000
sin(0.1)	0.099833
sin(0.2)	0.198669
sin(0.3)	0.295520
sin(0.4)	0.389418
sin(0.5)	0.479426
sin(0.6)	0.564642

Optimization: Larger and Faster (Loop Unrolling)

Loop unrolling reduces the overhead of jumping and condition testing by merging adjacent iterations.

Given a loop bound multiple of two

```
for (i = 0; i < 2 * N; i++) {
   a[i] = a[i] + b[i];
}
```

We can rewrite the code by merging pairs of iterations (unroll factor 2)

```
for (i = 0; i < 2 * N; i = i+2) {
   j = i + 1;
   a[i] = a[i] + b[i];
   a[j] = a[j] + b[j];
}
```

Loop unrolling can give a 10–20% speedup. What is a potential disadvantage? How does this work for loop bounds that may not be a multiple of the unroll factor?

For those interested, look into Duff's device

Optimizing High-Level Languages

The optimizer must undo fancy language abstractions

- Variables abstract away from registers, so the optimizer must find an efficient mapping;
- Control structures abstract away from gotos, so the optimizer must construct and simplify a goto graph;
- Data structures abstract away from memory, so the optimizer must find an efficient layout;

⋮

- Method lookups abstract away from procedure calls, so the optimizer must efficiently determine the intended implementations.

Optimizing High-Level Languages

The OO language BETA unifies as *patterns* the concepts

- Abstract class;
- Concrete class;
- Method; and
- Function.

A (hypothetical) optimizing BETA compiler must attempt to classify the patterns to recover that information.

Optimizing High-Level Languages

Difficult compromises

- A high abstraction level makes the development time cheaper, but the run-time more expensive; however
- High-level abstractions are also easier to analyze, which gives optimization potential.

Optimization considerations

- An optimizing compiler makes run-time more efficient, but compile-time less efficient;
- Optimizations for speed and size may conflict; and
- Different applications may require different optimizations.

JOOS Peephole Optimizer

- Works at the bytecode level;
- Looks only at *peepholes*, which are sliding windows on the code sequence;
- Uses *patterns* to identify and replace inefficient constructions;
- Continues until a global fixed point is reached; and
- Optimizes both speed and space.

JOOS Optimization

```
c = a * b + c;
if (c < a)
   a = a + b * 113;
while (b > 0) {
   a = a * c;
   b = b - 1;
}
```

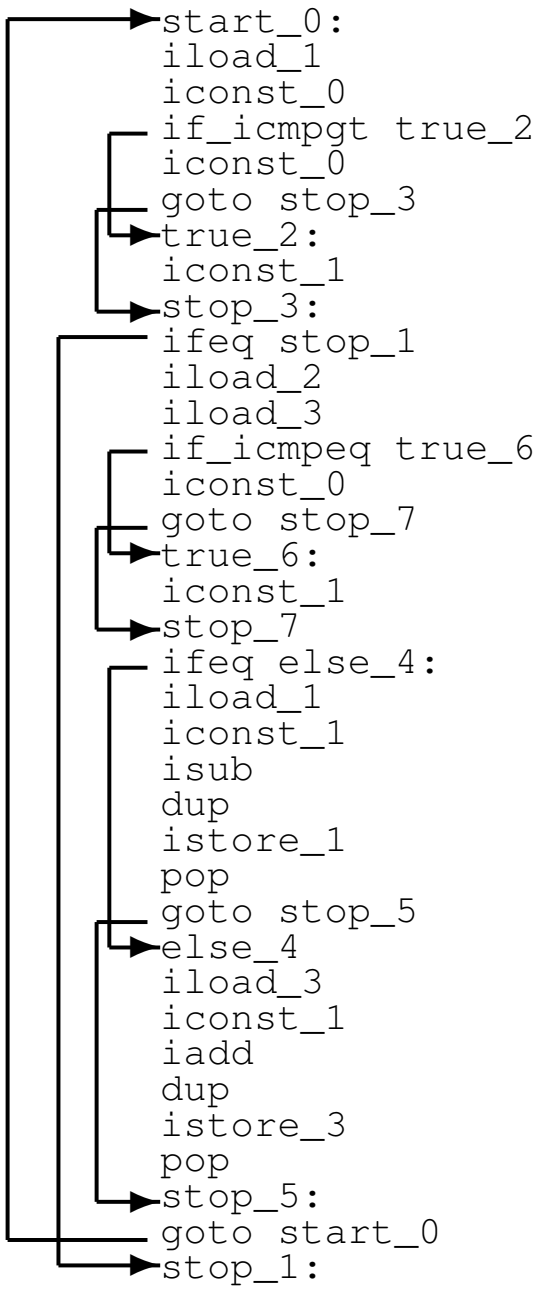
```
  iload_1
  iload_2
  imul
  iload_3
  iadd
  dup
  istore_3
  pop
  iload_3
  iload_1
  if_icmplt true_1
  iconst_0
  goto stop_2
true_1:
  iconst_1
stop_2:
  ifeq stop_0
  iload_1
  iload_2
  ldc 113
  imul
  iadd
  dup
  istore_1
  pop
stop_0:
start_3:
  iload_2
  iconst_0
  if_icmpgt true_5
  iconst_0
  goto stop_6
true_5:
  iconst_1
stop_6:
  ifeq stop_4
  iload_1
  iload_3
  imul
  dup
  istore_1
  pop
  ...
```

```
  iload_1
  iload_2
  imul
  iload_3
  iadd
  istore_3
  iload_3
  iload_1
  if_icmpge stop_0
  iload_1
  iload_2
  ldc 113
  imul
  iadd
  istore_1
stop_0:
start_3:
  iload_2
  iconst_0
  if_icmple stop_4
  iload_1
  iload_3
  imul
  istore_1
  iinc 2 -1
  goto start_3
stop_4:
```

Optimizer Goto Graph

The optimizer works on a structure called a goto graph that represents the jumps in a program

```
while (a > 0) {
   if (b == c)
      a = a - 1;
   else
      c = c + 1;
}
```



Optimizer Goto Graph

To capture the goto graph, the labels for a given code sequence are represented as an array of structures

```
typedef struct LABEL {
        char *name;
        int sources;
        struct CODE *position;
} LABEL;
```

Where

- The array index is the label's number;
- Field `name` is the textual part of the label;
- Field `sources` indicates the in-degree of the label; and
- Field `position` points to the location of the label in the code sequence.

Operations on the Goto Graph

The optimizer acts on the goto graph and may

- Inspect a given bytecode;
- Find the next bytecode in the sequence;
- Find the destination of a label;
- Create a new reference to a label;
- Drop a reference to a label;
- Ask if a label is dead (in-degree 0);
- Ask if a label is unique (in-degree 1); and
- **Replace a sequence of bytecodes by another**.

Optimizer - Instructions

A peephole optimizer can replace one sequence of instructions by another

- Traverse the bytecode sequence (`next`); and
- Check each instruction is in the pattern (`is_<inst>`).

Find the next bytecode in the sequence

```
CODE *next(CODE *c) {
   if (c == NULL) return NULL;
   return c->next;
}
```

Inspect a given bytecode

```
int is_istore(CODE *c, int *arg) {
   if (c == NULL) return 0;
   if (c->kind == istoreCK) {
      (*arg) = c->val.istoreC;
      return 1;
   } else {
      return 0;
   }
}
```

Optimizer - Labels

Optimizations may also traverse the goto graph and evaluate jump targets

Find the destination of a label

```
CODE *destination(int label) {
   return currentlabels[label].position;
}
```

Create a new reference to a label

```
int copylabel(int label) {
   currentlabels[label].sources++;
   return label;
}
```

Drop a reference to a label

```
void droplabel(int label) {
   currentlabels[label].sources--;
}
```

Optimizer - Labels

Optimizations may check properties of labels (for instance to remove dead labels)

Ask if a label is dead (in-degree 0)

```
int deadlabel(int label) {
   return currentlabels[label].sources == 0;
}
```

Ask if a label is unique (in-degree 1)

```
int uniquelabel(int label) {
   return currentlabels[label].sources == 1;
}
```

Optimization - Replace

When a peephole pattern identifies a sequence of bytecode to optimize, it can replace them by another

```
int replace(CODE **c, int k, CODE *r) {
   CODE *p = *c;
   for (int i = 0; i < k; i++) p = p->next;
   if (r == NULL) {
      *c = p;
   } else {
      *c = r;
      while (r->next != NULL) r = r->next;
      r->next = p;
   }
   return 1;
}
```

1. Find the first instruction that is not replaced (`i`);
2. Insert the new sequence (if there is one); and
3. Attach the end of the new sequence to instruction `i`.

Peephole Pattern - Positive Increment

An increment to a local variable

```
x = x + k
```

may be simplified to an increment operation, if $0 \leq k \leq 127$

Corresponding JOOS peephole pattern

```
int positive_increment(CODE **c) {
    int x, y, k;
    if (is_iload(*c, &x) &&
        is_ldc_int(next(*c), &k) &&
        is_iadd(next(next(*c))) &&
        is_istore(next(next(next(*c))), &y) &&
        x == y && 0 <= k && k <= 127) {
            return replace(c, 4, makeCODEiinc(x, k, NULL));
    }
    return 0;
}
```

We may attempt to apply this pattern anywhere in the code sequence.

Peephole Pattern - Algebraic Rules

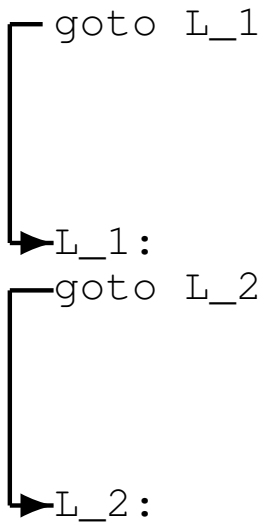
```
x * 0 = 0
x * 1 = x
x * 2 = x + x
```

Corresponding JOOS peephole pattern

```
int simplify_multiplication_right(CODE **c) {
    int x, k;
    if (is_iload(*c, &x) &&
        is_ldc_int(next(*c), &k) &&
        is_imul(next(next(*c)))) {
            if (k == 0)
                return replace(c, 3, makeCODEldc_int(0, NULL));
            else if (k == 1)
                return replace(c, 3, makeCODEiload(x, NULL));
            else if (k == 2)
                return replace(c, 3,
                    makeCODEiload(x, makeCODEdup(makeCODEiadd(NULL)))
                );
            return 0;
    }
    return 0;
}
```

Peephole Pattern - Goto Goto

A part of the goto graph may be simplified by short-circuiting the jump to `L_1`



Corresponding JOOS peephole pattern

```
int simplify_goto_goto(CODE **c) {
   int l1, l2;
   if (is_goto(*c, &l1) &&
      is_goto(next(destination(l1)), &l2) && l1 > l2) {
         droplabel(l1);
         copylabel(l2);
         return replace(c, 1, makeCODEgoto(l2, NULL));
   }
   return 0;
}
```

Peephole Pattern - Goto Goto

Why the condition `l1 > l2`?

```
int simplify_goto_goto(CODE **c) {
   int l1, l2;
   if (is_goto(*c, &l1) &&
      is_goto(next(destination(l1)), &l2) && l1 > l2) {
         droplabel(l1);
         copylabel(l2);
         return replace(c, 1, makeCODEgoto(l2, NULL));
   }
   return 0;
}
```

Consider the following bytecode

```
l1: goto l2
l2: goto l1
```

What will happen without this condition?

Announcements (Monday, February 26th)

Milestones

- Peephole out today! **Due:** Sunday, April 8th 11:59 PM
- Milestone 2 out today! **Due:** Sunday, March 11th 11:59 PM

Milestone 1

- Any last minute questions?
- **Due:** Tuesday, February 27th 11:59 PM

Peephole Pattern - Simplfy astore

The following JOOS peephole pattern removes an unnecessary `dup`/`pop` pair of instructions

```
int simplify_astore(CODE **c) {
   int x;
   if (is_dup(*c) &&
      is_astore(next(*c), &x) &&
      is_pop(next(next(*c)))) {
         return replace(c, 3, makeCODEastore(x, NULL));
   }
   return 0;
}
```

It is clearly sound, but will it ever be useful?

Peephole Pattern - Simplfy astore

Yes, the assignment expression statement

```
a = b;
```

We generate the assignment expression without the surrounding statement context - and therefore leave the value on the top of the stack

```
aload_2
dup
astore_1
pop
return
```

The final pop instruction is generated at the statement level

Peephole Pattern - Simplfy astore

The context agnostic generation for assignment expressions inserts the `dup` instruction by default

Corresponding JOOS source code

```
void codeEXP(EXP *e) {
   case assignK:
      codeEXP(e->val.assignE.right);
      code_dup();
      switch (e->val.assignE.leftsym->kind) {
         [...]
         case formalSym:
            if (e->val.assignE.leftsym->val.formalS->type->kind == refK) {
               code_astore(e->val.assignE.leftsym->val.formalS->offset);
            } else {
               code_istore(e->val.assignE.leftsym->val.formalS->offset);
            }
            break;
```

This handles chains of assignments `a = b = c` where the value is later needed

Peephole Pattern - Simplfy astore

To avoid the `dup` in the assign template

- We must know if the assigned value is needed later (contextual information); and
- It must also flow the decision back to the enclosing code below.

```
void codeSTATEMENT(STATEMENT *s) {
   case expK:
      codeEXP(s->val.expS);
      if (s->val.expS->type->kind != voidK) {
         code_pop();
      }
      break;
```

A peephole pattern is simpler and more modular.

Peephole Optimization

The peephole optimizer applies the collection of patterns in a fixed point process

```
repeat
   for each bytecode in succession do
      for each peephole pattern in succession do
         repeat
            apply the peephole pattern to the bytecode
         until the goto graph didn't change
      end
   end
until the goto graph didn't change
```

Peephole Optimization Termination

Why does this process terminate?

- Each peephole pattern does not necessarily make the code smaller; so
- To demonstrate termination for our examples, we use the lexicographically ordered measure

$$< \text{\#bytecodes}, \texttt{\#imul}, \sum_{\texttt{L}} |\mathit{gotochain}(\texttt{L})| >$$

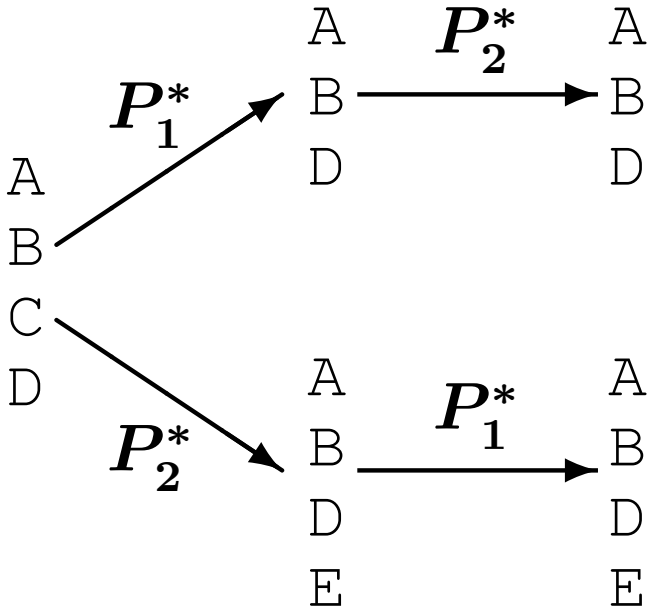
which can be seen to become strictly smaller after each application of a peephole pattern.

Peephole Optimization Fixed Point

- The goto graph obtained as a fixed point is *not* unique; since
- It depends on the sequence in which the peephole patterns are applied.

That does not happen for the four examples given, but consider the two peephole patterns

These patterns do not commute



JOOS Peephole Optimizer (patterns.h)

```
/* patterns here */

int simplify_astore(CODE **c)
{ int x;
  if (is_dup(*c) &&
      is_astore(next(*c), &x) &&
      is_pop(next(next(*c)))) {
    return replace(c, 3, makeCODEastore(x, NULL));
  }
  return 0;
}

[...]

int init_patterns() {
      ADD_PATTERN(simplify_multiplication_right);
      ADD_PATTERN(simplify_astore);
      ADD_PATTERN(positive_increment);
      ADD_PATTERN(simplify_goto_goto);
      return 1;
}
```

JOOS Peephole Optimizer (Fixed Point Driver)

```
int optiCHANGE;

void optiCODEtraverse(CODE **c) {
   int change = 1;
   if (*c != NULL) {
      while (change) {
         change = 0;
         for (int i = 0; i < OPTS; i++) {
            change = change | optimization[i](c);
         }
         optiCHANGE = optiCHANGE || change;
      }
      if (*c != NULL) optiCODEtraverse(&((*c)->next));
   }
}

void optiCODE(CODE **c) {
   optiCHANGE = 1;
   while (optiCHANGE) {
      optiCHANGE = 0;
      optiCODEtraverse(c);
   }
}
```

JOOS A+ Peephole Optimizer (40 peephole patterns)

Program	joosa+	joosa+ -O
AllComponents	907	861
AllEvents	1056	683
Animator	184	180
Animator2	568	456
ConsumeInteger	164	107
DemoFont	97	89
DemoFont2	213	147
DrawArcs	60	60
DrawPoly	94	90
Imagemap	470	361
MultiLineLabel	526	406
ProduceInteger	149	96
Rectangle2	58	58
ScrollableScribble	566	481
ShowColors	88	68
TicTacToe	1471	1211
YesNoDialog	315	248

“Optimizer”

The word “optimizer” is somewhat misleading, since the code is not optimal but merely “better”

Can we find the optimal?

Suppose *OPM*(G) is the shortest goto graph equivalent to G. The shortest diverging goto graph is

$$D_{\text{min}} \quad = \quad \begin{array}{l} \texttt{L:} \\ \texttt{goto L} \end{array}$$

We can then decide the Halting problem on an arbitrary goto graph G as

$$\textit{OPM}(G) = D_{\text{min}}$$

Hence, the program *OPM* cannot exist.

Testing

The testing strategy for the optimizer has three phases

1. A careful argumentation that each peephole pattern is sound;
2. A demonstration that each peephole pattern is realized correctly; and
3. A statistical analysis showing that the optimizer improves the generated programs.

There is a fine line between “optimization” and “not being stupid”

- R. Kent Dybvig