CS3300 - Language Translators

Introduction to Optimizations

V. Krishna Nandivada

IIT Madras

Introduction to optimizations

Optimization

Goal: produce fast code

- What is optimality?
- Problems are often hard
- Many are intractable or even undecideable
- Many are NP-complete
- Which optimizations should be used?
- Many optimizations overlap or interact

Optimization

Definition: An optimization is a transformation that is expected to:

improve the running time of a program
or decrease its space requirements

The point:

- "improved" code, not "optimal" code (forget "optimum")
- sometimes produces worse code
- range of speedup might be from 1.000001 to xxx

- It is undecidable whether in most cases, a particular optimization improves or (at least does not worsen) performance.
- Q: Can we not even say a simple transformation like algebraic simplification will always improve the code?

Optimization - conflicting goals

Typical goals of optimization for the generated code:

- Speed
- Space
- Power

Qs:

- Which one matters? depends on the target machine.
- Traditionally (hence the default behavior) compilers have targeted the speed of the generated code.
- Some times improving one goal improves another.
- Some times it does not. Example: loop unrolling, strength reduction (mult 5).

Choosing the optimizations

Some optimizations are more important than others.

Optimizations that apply to

- loops
- impact register allocation
- instruction scheduling

are essential for high performance.

Choice of optimizations may depend on the input program:

- OO programs - inlining (why ?) and leaf-routine optimizations.
- For recursive programs - tail call optimizations (replace some calls by jumps)
- For self-recursive programs - turn the recursive calls to loops.

Types of optimizations

Classification of optimization (based on their scope)

- Local (within basic blocks)
- Intra-procedural
- Inter-procedural

Classification based on their positioning:

- High level optimizations (use the program structure to optimize).
- Low level optimizations (work on medium/lower level IR)

Optimization classification (contd)

Classification with respect to their dependence on the target machine.

Machine independent

- applicable across broad range of machines
- move evaluation to a less frequently executed place
- specialize some general-purpose code
- remove redundant (unreachable, useless) code.
- create opportunities.

Machine dependent

- capitalize on machine-specific properties
- improve mapping from IR onto machine
- strength reduction.
- replace sequence of instructions with more powerful one (use “exotic” instructions)

A classical distinction

The distinction (Machine specific / independent) is not always clear: replace `multiply` with `shift`s and `add`s

Optimization

Desirable properties of an optimizing compiler

- code at least as good as an assembler programmer
- stable, robust performance (predictability)
- architectural strengths fully exploited
- architectural weaknesses fully hidden
- broad, efficient support for language features
- instantaneous compiles

Unfortunately, modern compilers often drop the ball

Types of program analysis

Classification of analysis (based on their view)

```
if (cond) {
   a = ...
   b = ...
} else {
   a = ...
   c = ...
}
// Which of the variables may be assigned? -- {a,b,c}
// Which of the variables must be assigned? -- {a}
```

- May analysis – the analysis holds on at least one data flow path.
- Must analysis – the analysis must hold on all data flow paths.

Example optimization: constant propagation

Goal:
Find the constant expressions in the program.

Replace all the constant expressions with their constant literals.

```
foo (int b) {
  a[1] = 1; a[2] = 2; a[3] = 3;
  i = 1;
  if (b > 2) {
    j = 2;
  } else {
    j = i + 1; }
  k = a[j];
  return k;
}
```

Classification of analysis (contd)

Classification of analysis (based on precision)

- Flow sensitive / insensitive.
  - Insensitive - the analysis should hold at every program point; does not depend on the type of control flow.
  - Sensitive - Each program point has its own analysis.

```
if (c) {
  a = 2;
  b = a;
  c = 3;
  print (a, b, c); // constants?
} else {
  a = 3
  b = a;
  c = 3;
  print (a, b, c); // constants?
}
```


Classification of analysis (contd)

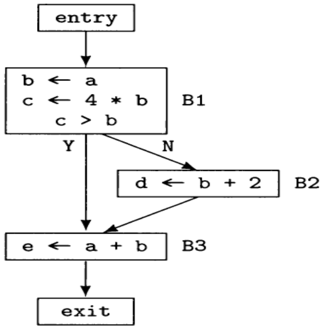
- Context sensitive and insensitive

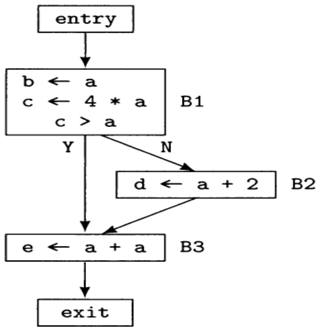
```
a = foo(2);

b = foo (3);

c = bar (2);

d = bar(2);

print (a, b, c, d); // a, b, c, d constants?

int foo(int x) { return x }
int bar(int x) { return x * x }
```


Copy propagation

Copy propagation: given an assignent `x := y`, replaces the later uses of `x` with `y`, provided that

the intervening instructions do not change the value of either `x` or `y`





Copy propagation (effect)

A seemingly simple and weak optimization.

Eliminates copy instructions.

Helps turns code into dead code.

Helps in register allocation (fewer live ranges)

Assists in other optimizations.

Can be done both localy (basic block level) or globaly (whole procedure).

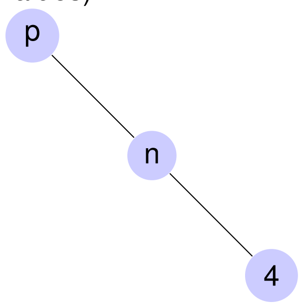
Alias Analysis

Alias analysis: problem of identifying storage locations that can be accessed by more than one way.

Are variable `a` and `b` aliases? ⇒ `a` and `b` refer to the same location? Modifying the contents of `a`, modifies the contents of `b`.

Unlike in copy propagation, in alias analysis we only talk about memory references (and not scalar values).

```
foo(){
  int *p;
  int n;
  p = &n;
  n = 4;
  print ("%d", *p);
}
```



Alias analysis (contd)

```
extern int *q;
foo( ) {
  int a, k;
  k = a + 5;
  f (a, &k);
  *q = 13;
  k = a + 5;   /* Assignment is redundant? */
               /* Expression is redundant? */
}
```

Only if a) the assignment to `*q` does not change `k` or `a`, b) the function call `f`, does not change `*k`.

Alias analysis (contd)

Granularity of analysis: Flow sensitive or insensitive.

```
bar ( ) {
  int a, b, e[], d, i;
  extern int *q;
  q = &a;
  a = 2;
  b = *q + 2;
  q = &b;
  for (i = 0; i < 100; i++) {
    e [i] = e [i] + a;
    *q = i;
  }
  d = *q + a;
}
```


Loop unrolling

(Example) Matrix-matrix multiply

```
do i ← 1, n, 1
   do j ← 1, n, 1
      c(i,j) ← 0
      do k ← 1, n, 1
         c(i,j) ← c(i,j) + a(i,k) * b(k,j)
```

- All the array elements are floating point values.
- $2n^3$ flops, n^3 loop increments and branches
- each iteration does 2 loads and 2 flops

This is the most overstudied example in the literature

Example: loop unrolling

Matrix-matrix multiply (assume 4-word cache line)

```
do i ← 1, n, 1
   do j ← 1, n, 1
      c(i,j) ← 0
      do k ← 1, n, 4
         c(i,j) ← c(i,j) + a(i,k) * b(k,j)
         c(i,j) ← c(i,j) + a(i,k+1) * b(k+1,j)
         c(i,j) ← c(i,j) + a(i,k+2) * b(k+2,j)
         c(i,j) ← c(i,j) + a(i,k+3) * b(k+3,j)
```

- $2n^3$ flops, $\frac{n^3}{4}$ loop increments and branches
- each iteration does 8 loads and 8 flops
- memory traffic is better
  - `c(i,j)` is reused (put it in a register)
  - `a(i,k)` reference are from cache
  - `b(k,j)` is problematic

Example: loop unrolling

Matrix-matrix multiply (to improve traffic on `b`)

```
do j ← 1, n, 1
   do i ← 1, n, 4
      c(i,j) ← 0
      do k ← 1, n, 4
         c(i,j) ← c(i,j) + a(i,k) * b(k,j)
            + a(i,k+1) * b(k+1,j) + a(i,k+2) * b(k+2,j)
            + a(i,k+3) * b(k+3,j)
         c(i+1,j) ← c(i+1,j) + a(i+1,k) * b(k,j)
            + a(i+1,k+1) * b(k+1,j)
            + a(i+1,k+2) * b(k+2,j)
            + a(i+1,k+3) * b(k+3,j)
         c(i+2,j) ← c(i+2,j) + a(i+2,k) * b(k,j)
            + a(i+2,k+1) * b(k+1,j)
            + a(i+2,k+2) * b(k+2,j)
            + a(i+2,k+3) * b(k+3,j)
         c(i+3,j) ← c(i+3,j) + a(i+3,k) * b(k,j)
            + a(i+3,k+1) * b(k+1,j)
            + a(i+3,k+2) * b(k+2,j)
            + a(i+3,k+3) * b(k+3,j)
```


Example: loop unrolling

What happened?

- interchanged `i` and `j` loops
- unrolled `i` loop
- fused inner loops
- $2n^3$ flops, $\frac{n^3}{16}$ loop increments and branches
- first assignment does 8 loads and 8 flops
- 2nd through 4th do 4 loads and 8 flops
- memory traffic is better
  - `c(i,j)` is reused (register)
  - `a(i,k)` references are from cache
  - `b(k,j)` is reused (register)

Loop optimizations: factoring loop-invariants

Loop invariants: expressions constant within loop body

Goal: move the loop invariant computation to outside the loop.

The loop independent code executes only once, instead of many times the loop might.

Example: loop invariants

```
foreach i=1 .. 100 do
  foreach j=1 .. 100 do
    foreach k=1 .. 100 do
      A[i,j,k] = i * j * k;
    end
  end
end
```

- 3 million index operations
- 2 million multiplications

Example: loop invariants (cont.)

Factoring the inner loop:

```
foreach i=1 .. 100 do
  foreach j=1 .. 100 do
    t1 = &A[i][j];
    t2 = i * j ;
    foreach k=1 .. 100 do
      t1[k] = t * k;
    end
  end
end
```

And the second loop:

```
foreach i=1 .. 100 do
  t3 = &A[i];
  foreach j=1 .. 100 do
    t1 = &t3[j];
    t2 = i * j ;
    foreach k=1 .. 100 do
      t1[k] = t * k;
    end
  end
end
```


Instruction Scheduling

Instruction scheduling

Optimization - overview

Good compilers are crafted, not assembled

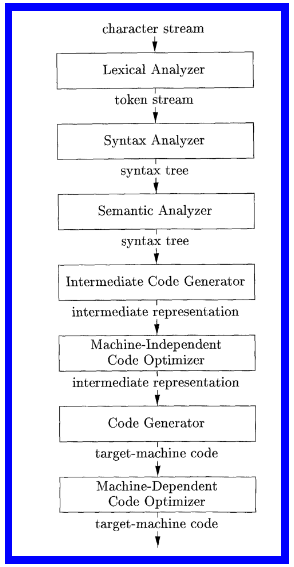
- consistent philosophy
- careful selection of transformations
- thorough application
- coordinate transformations and data structures
- attention to results (code, time, space)

Compilers are engineered objects

- minimize running time of compiled code
- minimize compile time
- use reasonable compile-time space (serious problem)

Thus, results are sometimes unexpected

Back to first lecture



Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

Back end responsibilities:

- Optimizations, code generation.

Our target

- five out of seven phases.
- glance over optimizations – attend the graduate course, if interested.