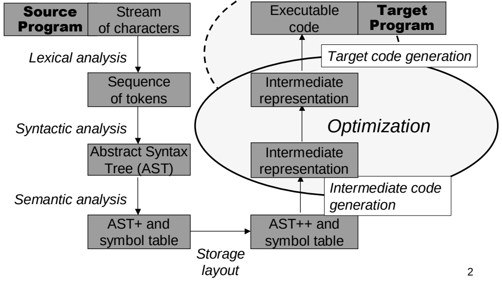
CSE401: Optimization

Larry Snyder

Autumn 2003

Slides by Chambers, Eggers, Notkin, Ruzzo, Snyder and others

Prototype compiler structure



Optimization

- Identify inefficiencies in target or intermediate code
- Replace with equivalent but “better” sequences
- “Optimize” is a lie.
  “Usually improve” is more honest.

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] := …
```

Kinds of optimizations

- Scope of analysis is central to what optimizations can be performed. A larger scope may expose better optimizations, but is more complex
  - *Peephole*: look at adjacent instructions
  - *Local*: look at straight-line sequences of instructions
  - *Global (intraprocedural)*: look at whole procedure
  - *Interprocedural*: look across procedures

Increasing scope, opportunity, and complexity

Peephole

- After codegen, look at a few adjacent instructions
  - Try to replace them with something better
- If you have
```
sw $8,12($fp)
lw $12,12($fp)
```
- You can replace it with
```
sw $8,12($fp)
mv $12,$8
```

Peephole examples: 68k

If you have | Replace it with

```
sub sp,4,sp      }    mov r1,-(sp)
mov r1,0(sp)     }
```

```
mov 12(fp),r1    }
add r1,1,r1      }    inc 12(fp)
mov r1,12(fp)    }
```

Peephole optimization of jumps

- Eliminate
  - Jumps to jumps
  - Conditional branch over unconditional branch
- “Adjacent instructions” means “adjacent in control flow”

```
if a < b then
  if c < d then
    # do nothing
  else
    stmt1;
  end;
else
  stmt2;
end;
```

```
   if (a≥b)goto 1
   if (c≥d)goto 2
   #do nothing
   goto 3
2:stmt1
3:
   goto 4
1:stmt2
4:
```

How to do peephole opts

- Could be done at IR and/or target level
- Catalog of specific code rewrite templates
- Scan code with moving window looking for matches

Peephole summary

- You could consider peephole optimization as increasing the sophistication of instruction selection
- Relatively easy to do
- Relatively easy to extend
- Relatively easy to ensure correctness
- Relatively high payoff

Algebraic simplifications
by peephole or codegen

- “constant folding” and “strength reduction” are common names for this kind of optimization
  - `z := 3 + 4`
  - `z := x + 0`
`z := x * 1`
  - `z := x * 2`
`z := x * 8`
`z := x / 8`
  - `float x,y;`
`z := (x + y) – y;`

Local optimization

- Analysis and optimizations within a basic block

A *basic block* is a straight-line sequence of statements with no control flow into or out of the middle of the sequence

- Local optimizations are more powerful than peephole (e.g., block may be longer than peephole window)
  - Not too hard to implement
  - Can be machine-independent, if done on intermediate code

Local constant propagation (aka "constant folding")

- If a constant is assigned to a variable, replace downstream uses of the variable with the constant
- If all operands are const, replace with result
- May enable further constant folding

Example

```
const count : int = 10;
…
x := count * 5;
y := x ^ 3;
```

```
t1  := 10
t2  := 5
t3  := t1 * t2
x   := t3

t4  := x
t5  := 3
t6  := exp(t4,t5)
y   := t6
```

Local dead assignment elimination

- If the left hand side of an assignment is never read again before being overwritten, then remove the assignment

- This sometimes happens while cleaning up from other optimizations (as with many of the optimizations we consider)

Example

```
const count : int = 10;
…
x := count * 5;
y := x ^ 3;
x := input;
```

```
x  := 50
t6 := exp(50,3)
y  := t6
x  := input()
```

Intermediate code after constant propagation

Common subexpression elimination

- Avoid repeating the same calculation
- Requires keeping track of available expressions

CSE example: `… 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
```

Next

- Intraprocedural optimizations
  - Code motion
  - Loop induction variable elimination
  - Global register allocation
- Interprocedural optimizations
  - Inlining
- After that…how to implement these optimizations
- ∃ other kinds of optimizations, beyond the scope of this class, e.g. dynamic compilation

Int*ra*procedural optimizations

- Enlarge scope of analysis to entire procedure
  - Provides more opportunities for optimization
  - Have to deal with branches, merges and loops
- Can do constant propagation, common subexpression elimination, etc. at this level
- Can do new things, too, like loop optimizations
- Optimizing compilers usually work at this level

Code motion

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

```
for i := 1 to 10 do
  a[i] := a[i] + b[j];
  z := z + 10000
end
```

At intermediate code level

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

```
  *(fp+ioffset) := 1
_l0:
  if *(fp+ioffset) > 10 goto _l1
  t1 := *(fp+joffset)
  t2 := t1*4
  t3 := fp+t2
  t4 := *(t3+boffset)
  t5 := *(fp+ioffset)
  t6 := t5*4
  t7 := fp+t6
  *(t7+aoffset) := t4
  t8 := *(fp+ioffset)
  t9 := t8+1
  *(fp+ioffset) := t9
  goto _l0
_l1:
```

Loop induction variable elimination

- For-loop index is an *induction variable*
  - Incremented each time through the 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 the loop

Example

```
for i := 1 to 10 do        for p := &a[1] to &a[10] do
  a[i] := a[i] + x;          *p := *p + x;
end                        end
```

Global register allocation

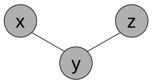
- Try to allocate local variables to registers
- If two locals don’t overlap, then give them the same register
- Try to allocate most frequently used variables to registers first

```
proc f(n:int,x:int):int;
  var sum: int, i:int;
begin
  sum := x;
  for i := 1 to n do
    sum := sum + i;
  end
  return sum;
end f;
```

Register allocation by coloring

- As before, IR gen as if infinite regs avail
- Build *interference graph:*

```
x := a+5;
y := b*2;
z := x/3;
a := y-2;
```



- Colorable with few colors (regs)?
  - NP-hard, but …
- If not, pick a node & generate spill code

Interprocedural optimizations

- What happens if we expand the scope of the optimizer to include procedures calling each other
  - In the broadest scope, this is optimization of the program as a whole
- We can do local, intraprocedural optimizations at a bigger scope
  - For example, constant propagation
- But we can also do entirely new optimizations, such as inlining

Interprocedural opt: Issues

```
procedure P() {
  x: int;
  x := 10;
  Q(          );
  x:= x+1;
  if x == 11 then
  …
```

- Q()
- Q(x by value)
- Q(x by reference)
- Q(const x by reference)
- Q(), but Q declared in P
- …

Inlining

Replace procedure call with the body of the called procedure

```
const pi:real := 3.14159;
proc area(rad:int):int;
begin
  return pi*(rad^2);
end;
…
r := 5;
…
output := area(r);
```

```
const pi:real := 3.14159;
proc area(rad:int):int;
begin
  return pi*(rad^2);
end;
…
r := 5;
…
output := pi*(r^2);
```

Questions about inlining:
few answers

- How to decide where the payoff is sufficient to inline?
  - The real decision depends on dynamic information about frequency of calls
- In most cases, inlining causes the code size to increase; when is this acceptable?
- Others?

Optimization and debugging

- Debugging optimized code is often hard
- For example, what if:
  - Source code statements have been reordered?
  - Source code variables have been eliminated?
  - Code is inlined?
- In general, the more optimization there is, the more complex the back-mapping is from the target code to the source code … which can confuse a programmer

Summary of optimization

- Larger scope of analysis yields better results
  - Most of today's optimizing compilers work at the intraprocedural level, with some doing some work at the interprocedural level
- Optimizations are usually organized as collections of passes
- The presence of optimizations may make other parts of the compiler (e.g., code gen) easier to write
  - E.g., use a simple instruction selection algorithm, knowing that the optimizer can, in essence, act to improve these instruction selections

Implementing intraprocedural optimizations

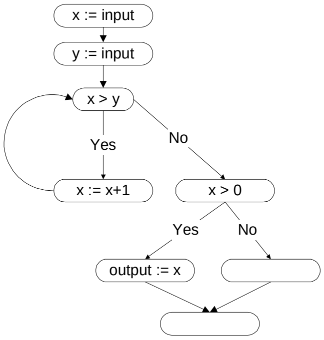
- The heart of implementing optimizations is the definition and construction of a convenient representation
- We'll look a bit more closely at two common and useful representations
  - The control flow graph (CFG)
  - The data flow graph (DFG)

CFG

- Nodes are intermediate language statements
  - Or whole basic blocks
- Edges represent control flow
- Node with multiple successors is a branch/switch
- Node with multiple predecessors is a merge
- Loop in a graph represents a loop in the program

Example

```
while x > y do
  x : = x + 1;
end;
if x > 0 then
  output := x;
end;
```

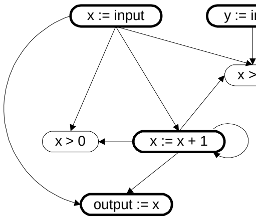


DFG: def/use chains

- Nodes are def(initions) and uses
- Edge from def to use
- A def can reach multiple uses
- A use can have multiple reaching defs

Example

```
x := input;
y := input;
while x > y do
  x : = x + 1;
end;
if x > 0 then
  output := x;
end;
```



Example program
CFG and DFG in groups

```
x := 3;
y := x * x;
if y > 10 then
  x := 5;
  y := y + 1;
else
  x := 6;
  y := x + 4;
end;
w := y / 3;
while y > 0 do
  z := w * w;
  x := x – z;
  y := y – 1;
end;
output  := x;
```

Analysis and transformation

- Each optimization is one or more analyses followed by a transformation
- Analyze CFG and/or DFG by propagating information forward or backward along CFG and/or DFG edges
  - Merges in graph require combining information
  - Loops in graph require iterative approximation
- Perform improving transformations based on information computed
  - Have to wait until any iterative approximation has converged
- Analysis must be conservative, so that transformations preserve program behavior

A simple analysis

- Let's start with a simple analysis that can help us determine which assignments can be eliminated from a basic block
- The example is unreasonable as source, but perhaps not as intermediate code

```
proc foo(j, k,
l:int):int
begin
  int a, b, c, n, x;
  a := 17 * j;
  b := k * k;
  c := a + b;
  a := k * 7;
  return c;
end
```

Liveness analysis

- This analysis is a form of liveness analysis
  - It can help identify assignments to remove
  - It can also form the basis for memory and register optimizations
- The goal is to identify which variables are *live* and which are *dead* at given program points
- The analysis is usually performed backwards
  - When a variable is used, it becomes lives in that statement and code before it
  - When a variable is assigned to, it becomes dead for all code before it
- Note the relationship to def-use, as we saw in the data flow graph

Work backwards

	Live	Dead
`proc foo(j, k, l:int):int`		
`begin`		
`int a, b, c, n, x;`		
`a := 17 * j;`	?	?
`b := k * k;`	?	?
`c := a + b;`	{k,l,a,b,c}	{j,n,x}
`a := k * l;`	{k,l,c}	{j,n,x,a,b}
`return c;`	{c}	{j,k,l,n, x,a,b}
`end`		

So?

- This analysis shows we can eliminate the last assignment to a, which is no surprise
- Technically, assignments to a dead variable can be removed
  - The value isn't needed below, so why do the assignment?
- Furthermore, you could show for this example that the declarations for n and x aren't needed, since n nor x is ever live

Then…

- After eliminating the last assignment (and these two declarations), you can redo the analysis
- This analysis now shows that `l` is dead everywhere in the block, and it can be removed as a parameter
- The stack can be reduced because of this
- And the caller could, in principle, be further optimized

Well, that was easy

- But that's for basic blocks
- Once we have control flow, it's much harder to do because we don't know the order in which the basic blocks will execute
- We need to ensure (for optimization) that every possible path is accounted for, since we must make conservative assumptions to guarantee that the optimized code always works