



COMP 181


Lecture 16
Introduction to optimization

November 9, 2005

Project stage 3


- Original version:
 - "Allocation": walking the AST and assigning activation record offsets to each variable
 - "Tuple generation": walk the AST and generate low-level IR
- Modified version:
 - I'll give you the allocation part
 - Tuple generation
 - Expressions are done
 - Four kinds of stmts: call, case, for, repeat



2

Project stage 3


- Goal:
 - Focus on the ideas
 - Don't spend too much time learning minutia
- How?
 - I need to edit the files myself
 - Produce a new document with extra information
 - That takes time – might be today or tomorrow



3

Midterm


- Extra credit
 - $num \rightarrow 11 \mid 1001 \mid num \ 0 \mid num \ num$
- Induction:
 - Case 1: $11 = 3$ is divisible by 3
 - Case 2: $1001 = 9$ is divisible by 3
 - Case 3: $num \ 0$
 - Assuming num is divisible by 3
 - $num \ 0 = num * 2$ is divisible by 3



4

Midterm


- Induction continued
 - Case 4: $num \ num$
 - $num_0 \ num_1 = (num_0 * 2^n) + num_1$
 - Assuming both numbers divisible by 3
 - $num_0 = 3 * x$
 - $num_1 = 3 * y$
 - $num_0 \ num_1 = (3 * x * 2^n) + 3 * y$
 - Factor out the 3
 - $num_0 \ num_1 = 3 * (x * 2^n + y)$ is divisible by 3
- QED



5

Midterm

- Does this produce all numbers divisible by 3?
 - If true, proof would be difficult
 - If false,
 - Provide a counterexample: $10101 = 21$
 - Just to be a stickler: prove that 10101 cannot occur
 - Induction on number of 1 bits
 - Number of 1 bits is always even
 - What if we added $num \rightarrow 10101 ?$
- Extra, extra credit
 - "Scooter" – read the newspaper, people



6

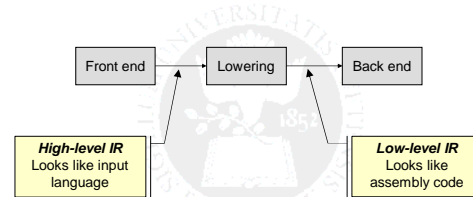
Prelude

- What is this?
The medal that accompanies a Nobel Prize
- Why did Alfred Nobel create the prize?
Apparently, guilt over his invention of dynamite
- What are the five Nobel Prize categories?
Physics, chemistry, medicine, literature, and peace
- What about math and computer science?
Fields medal, Turing award



Overview

- Where are we?



Back end

- At this point we could generate machine code
 - Output of lowering is a correct translation
 - What's left to do?
 - Map from lower-level IR to machine code
 - Maybe some register management (*could be required*)
 - Pass off to assembler
- Why have a separate assembler?
 - Handles "packing the bits"

Assembly	<code>addi <target>, <source>, <value></code>
Machine	<code>0010 00ss ssst tttt iiii iiii iiii iiii</code>



But first...

- The compiler "understands" the program
 - IR captures program semantics
 - Lowering: semantics-preserving transformation
 - Why not do others?
- Compiler optimizations
 - Oh great, now my program will be optimal!
 - Sorry, it's a misnomer
 - What is an "optimization"?



Optimizations

- What are they?
 - Code transformations
 - Improve some metric
- Metrics
 - Performance: time, instructions, cycles
 - Space
 - Reduce memory usage
 - Code Size
 - Energy



Why optimize?

- High-level constructs may make some optimizations difficult or impossible:
$$A[i][j] = A[i][j-1] + 1$$
$$t = A + i*row + j$$
$$s = A + i*row + j - 1$$
$$(*t) = (*s) + 1$$
- High-level code may be more desirable
 - Program at high level
 - Focus on design; clean, modular implementation
 - Let compiler worry about gory details
- Premature optimization is the root of all evil!

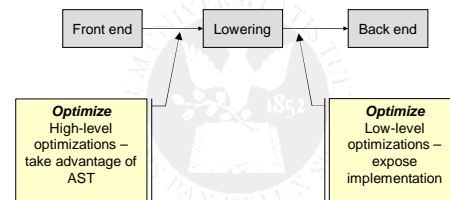


Limitations

- What are optimizers good at?
 - Consistency
 - Find all opportunities for an optimization
 - Uniformly apply the transformation
- What are they *not* good at?
 - Asymptotic complexity
 - Compilers can't fix bad algorithms
 - Compilers can't fix bad data structures
- There's no magic



Overview



Requirements

- Safety
 - Preserve the semantics of the program
 - What does that mean?
- Profitability
 - Will it help our metric?
- Risk
 - How will interact with other optimizations?
 - How will it affect other stages of compilation?



Example

- Loop unrolling
(We saw this in Duff's Device)
- Safety:
 - Always safe; getting loop conditions right can be tricky.
- Profitability
 - Depends on hardware – usually a win
- Risk
 - Increases size of code in loop
 - May not fit in the instruction cache



Optimizations

- Many, many optimizations invented
 - *Constant folding, constant propagation, tail-call elimination, redundancy elimination, dead code elimination, loop-invariant code motion, loop splitting, loop fusion, strength reduction, array scalarization, inlining, cloning, data prefetching, parallelization. . .etc . .*
- How do they interact?
 - Optimist: we get the sum of all improvements!
 - Realist: many are in direct opposition



Categories

- Traditional optimizations
 - Transform the program to reduce work
 - Don't change the level of abstraction
- Enabling transformations
 - Don't necessarily improve code on their own
 - Inlining, loop unrolling
- Resource allocation
 - Map program to specific hardware properties
 - Register allocation
 - Instruction scheduling, parallelism
 - Data streaming, prefetching




Constant propagation

- **Idea**

- If the value of a variable is known to be a constant at compile-time, replace the use of variable with constant

```
n = 10;
c = 2;
for (i=0; i<n; i++)
  s = s + i*c;
```



```
n = 10;
c = 2;
for (i=0; i<10; i++)
  s = s + i*2;
```

- **Safety**

- Prove the value is constant

- **Notice:**

- May interact *favorably* with other optimizations, like loop unrolling – now we know the *trip count*



Constant folding

- **Idea**

- If operands are known at compile-time, evaluate expression at compile-time

```
r = 3.141 * 10;
```



```
r = 31.41;
```

- **Often repeated throughout compiler**

```
x = A[2];
```



```
t1 = 2*4;
t2 = A + t1;
x = *t2;
```



Partial evaluation

- Constant propagation and folding together

- **Idea:**

- Evaluate as much of the program at compile-time as possible
- More sophisticated schemes:
 - Build arrays
 - Simulate data structures

- **Caveat: floating point**

- Preserving the error characteristics of floating point values




Algebraic simplification

- **Idea:**

- Apply the usual algebraic rules to simplify expressions

```
a * 1
a / 1
a * 0
a + 0
b || false
```



```
a
a
0
a
b
```

- Repeatedly apply to complex expressions

- Many, many possible rules
 - Associativity and commutativity come into play



Copy propagation

- **Idea:**

- After an assignment $x = y$, replace any uses of x with y

```
x = y;
if (x>1)
  s = x+f(x);
```



```
x = y;
if (y>1)
  s = y+f(y);
```

- **Safety:**

- Only apply up to another assignment to x

- What if there was an assignment $y = z$ earlier?

- Apply transitively to all assignments



Dead code elimination

- **Idea:**

- If the result of a computation is never used, then we can remove the computation

```
x = y + 1;
y = 1;
x = 2 * z;
```



```
y = 1;
x = 2 * z;
```

- **Safety**

- Variable is dead if it is never used after defined
- Remove code that assigns to dead variables

- This may, in turn, create more dead code

- Many other passes leave dead code



Common sub-expression elimination

- Idea:
 - If program computes the same expression multiple times, reuse the value.

```

a = b + c;
c = b + c;
d = b + c;
    
```

→

```

a = b + c;
c = a;
d = b + c;
    
```

- Safety:
 - Subexpression can only be reused until operands are redefined
- Often occurs in address computations
 - Array indexing and struct/field accesses



How do these things happen?

- Who would write code with:
 - Dead code
 - Common subexpressions
 - Constant expressions
 - Copies of variables
- Two ways they occur
 - High-level constructs – we've already seen examples
 - Other optimizations
 - Copy propagation often leaves dead code
 - Enabling transformations: inlining, loop unrolling, etc.



Unreachable code elimination

- Idea:
 - Eliminate code that can never be executed

```

#define DEBUG 0
. . .
if (DEBUG)
    print("Current value = ", v);
    
```

- Different implementations
 - High-level: look for if (false) or while (false)
 - Low-level: more difficult
 - Code is just labels and gotos
 - Traverse the graph, marking reachable blocks



Loop optimizations

- Program hot-spots are usually in loops
 - Most programs: 90% of execution time is in loops
 - What are possible exceptions?
 - OS kernels, compilers and interpreters
- Loops are a good place to expend extra effort
 - Numerous loop optimizations
 - For languages like Fortran, very effective
 - Many are more expensive optimizations



Loop-invariant code motion

- Idea:
 - If a computation won't change from one loop iteration to the next, move it outside the loop

```

for (i=0; i<N; i++)
    A[i] = A[i] + x*x;
    
```

→

```

t1 = x*x;
for (i=0; i<N; i++)
    A[i] = A[i] + t1;
    
```

- Safety:
 - Determine when expressions are invariant
- Useful for array address computations
 - Not visible at source level



Strength reduction

- Idea:
 - Replace expensive operations (multiplication, division) with cheaper ones (addition, subtraction, bit shift)
- Traditionally applied to induction variables
 - Variables whose value depends linearly on loop count
 - Special analysis to find such variables

```

for (i=0; i<N; i++)
    v = 4*i;
    A[v] = . . .
    
```

→

```


v = -4;
for (i=0; i<N; i++)
    v = v + 4;
    A[v] = . . .
    
```



Strength reduction

- Can also be applied to simple arithmetic operations:

```
x * 2  
x * 2^c  
x/2^c
```



```
x + x  
x<<c  
x>>c
```

- Typical example of premature optimization
 - Programmers use bit-shift instead of multiplication
 - “ $x << 2$ ” is harder to understand
 - Most compilers will get it right automatically



Loop unrolling

- Okay, enough with this example...



Inlining

- Idea:
 - Replace a function call with the body of the callee
- Safety
 - What about recursion?
- Risk
 - Code size
 - Most compilers use heuristics to decide when
 - Has been cast as a *knapsack problem*
- Critical for OO languages
 - Methods are often small
 - Encapsulation, modularity force code apart



Big picture

- When do we apply these optimizations?
 - High-level:
 - Inlining, cloning
 - Some algebraic simplifications
 - Low-level
 - Everything else
- It's a black art
 - Ordering is often arbitrary
 - Many compilers just repeat the optimization passes over and over



Summary

- Myriad optimizations to improve programs – particularly runtime
- Optimizations interact in both positive and negative ways
- Primary issue: safety



Next time...

- Project stage 3 – still in the works
- A lecture from Noah?
- When I get back: program analysis

