COMP 181

Lecture 16

Introduction to optimization

November 2, 2006

Prelude

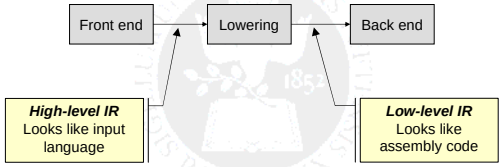
- What movie is this?
  - Terminator 2
- What's going on here?
  - T-1000 terminator can alter its shape, appearance
  - "Programmable matter"
  - Science fiction?

Claytronics

- Project at CMU
  - The goal of the claytronics project is to understand and develop the hardware and software necessary to create a **material** which can be **programmed** to form dynamic three dimensional **shapes** which can **interact** in the physical world and visually take on an arbitrary **appearance**
- "catoms"
  - Move in three dimensions (in relation to other catoms)
  - Adhere to other catoms to maintain a 3D shape
  - Compute state information
- Demo…

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	`addi <target>, <source>, <value>`
Machine	`0010 00ss ssst tttt iiii iiii iiii iiii`

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?
  - Consistent and thorough
  - 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

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:
    - Simulate data structures, arrays
    - Symbolic execution of the code
- 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

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
  - Dead-code elimination usually work transitively

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;
```

⇨

```
t = b + c
a = t;
c = t;
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.

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, **or**
  - …another assignment to y!
- What if there was an assignment y = z earlier?
  - Apply transitively to all assignments

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


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

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

Inlining

- In C:
  - At a call-site, decide whether or not to inline
    - (Often a heuristic about callee/caller size)
  - Look up the callee
  - Replace call with body of callee
- What about Java?
  - What complicates this?
  - Virtual methods
  - Even worse?
  - Dynamic class loading

```
class A { void M() {…} }
class B extends A
        { void M() {…} }

void foo(A x)
{
  x.M(); // which M?
}
```


Inlining in Java

- With guards:

```
void foo(A x)
{
  if (x is type A)
    x.M(); // inline A's M
  if (x is type B)
    x.M(); // inline B's M
}
```

- Specialization
  - At a given call, we may be able to determine the type
  - Requires fancy analysis

```
y = new A();
foo(y);
z = new B();
foo(z);
```


Control-flow simplification

- High-level optimization
- Idea:
  - If we know the value of a branch condition, eliminate the unused branch
- How would that happen?
  - Combination of other opts:
    - Constant propagation, constant folding
- What's the benefit?
  - Straight-line code
  - Easier to reason about, easier to optimize
  - Better for pipelined architectures

```
if (10 > 5) {
  ...
} else {
  ...
}
```

Anatomy of an optimization

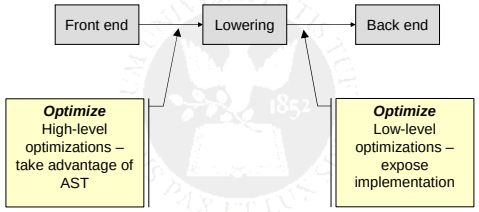
Two big parts:

- Program analysis
  Pass over code to find:
  - Opportunities
  - Satisfy safety constraints
- Program transformation
  - Change the code to exploit opportunity
- Often: rinse and repeat

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

Overview



Writing fast programs

In practice:

- Pick the right algorithms and data structures
  - Asymptotic complexity
  - Memory usage, indirection, representation
- Turn on optimization and profile
  - Run-time
  - Program counters (e.g., cache misses)
- Evaluate problems
- Tweak source code
  - Make the optimizer do "the right thing"

Summary

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

Next time…

- Program analysis
- Dead-code elimination