Code Optimization Techniques

Course Name: Compiler Design
Course Code: CSE331
Level:3, Term:3
Department of Computer Science and Engineering
Daffodil International University

The Back End

- At this point we could generate machine code
  - What's left to do?
    - Map from lower-level IR to machine code
    - Register management
    - Pass off to assembler

- Why have a separate assembler?
  - Handles "packing the bits"

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

But First…

- The compiler “understands” the program
  - IR captures program semantics
  - Lowering: semantics-preserving transformation
- Compiler optimizations
  - Now my program will be optimal!
    - Does it make best use of computing resources
  - What is an “optimization”?

Optimizations

- What are they?
  - Code transformations with preserved semantics
  - Improve some metric

- Metrics
  - Performance: time, instructions, cycles
  - Space: Reduce memory usage
  - Code Size
  - Energy

Optimizations

- What are they?

◆ Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

◆ In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

Optimizations

A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Optimizations Cont’d

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

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?
  - Being consistent and thorough
  - Find all opportunities for an optimization
  - Uniformly apply the transformation
- What are they *not* good at?
  - Asymptotic complexity (time analysis /Big O)
  - 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
- Profitability
  - Will it help our metric?
- Risk
  - How will interact with other optimizations?
  - How will it affect other stages of compilation?

Example: Loop Unrolling

- 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;
```

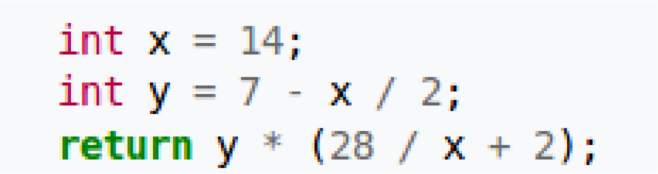
```
n = 10;
c = 2;
for (i=0;i<10;i++)
  s = s + i*2;
```

- Safety
  - Prove the value is constant
- Notice:
  - May interact favorably with other optimizations, like loop unrolling – now we know the *trip count*

Constant Folding

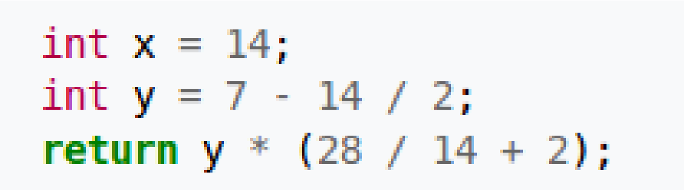
- <u>Idea</u>
  - If operands are known at compile-time, evaluate expression at compile-time

`r = 3.141 * 10;` → `r = 31.41;`



```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

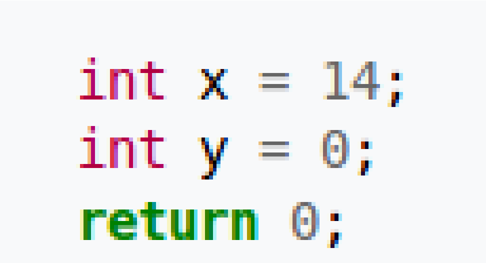
- Propagating x yields:



```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Constant Folding Cont’d

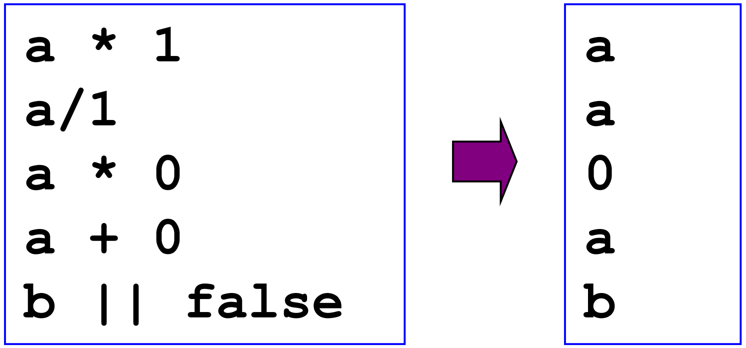
- Continuing to propagate yields the following (which would likely be further optimized by dead code elimination of both x and y)



```
int x = 14;
int y = 0;
return 0;
```

Algebraic Simplification

- <u>Idea</u>:
  - 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

- Dead code is one or more than one code statements, which are:

◆ Either never executed or unreachable,
◆ Or if executed, their output is never used.

- Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

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 works 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

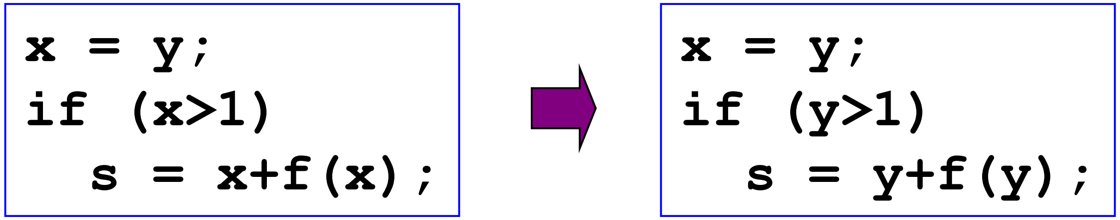
How Do These Things Happen?

- Who would write code with:
  - Dead code
  - Common subexpressions
  - Constant expressions
  - Copies of variables

- Two ways they occur
  - High-level constructs – already saw 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 were 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
  - 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 (mult, div) with cheaper ones (add, sub, bit shift)

- Traditionally applied to induction variables
  - Variables whose value depends linearly on loop count
  - Special analysis to find such variables

Strength Reduction

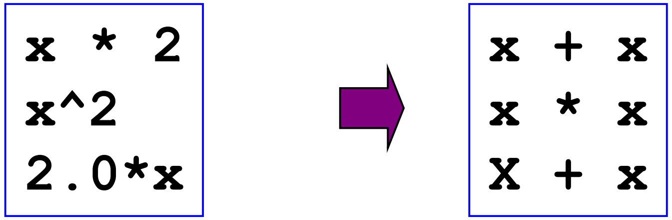
```
for (i=0;i<N;i++)
  v = 4*i;
  A[v] = . . .
```

➡

```
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
2.0*x
```

```
x + x
x * x
X + x
```

  - This improves execution time
- 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

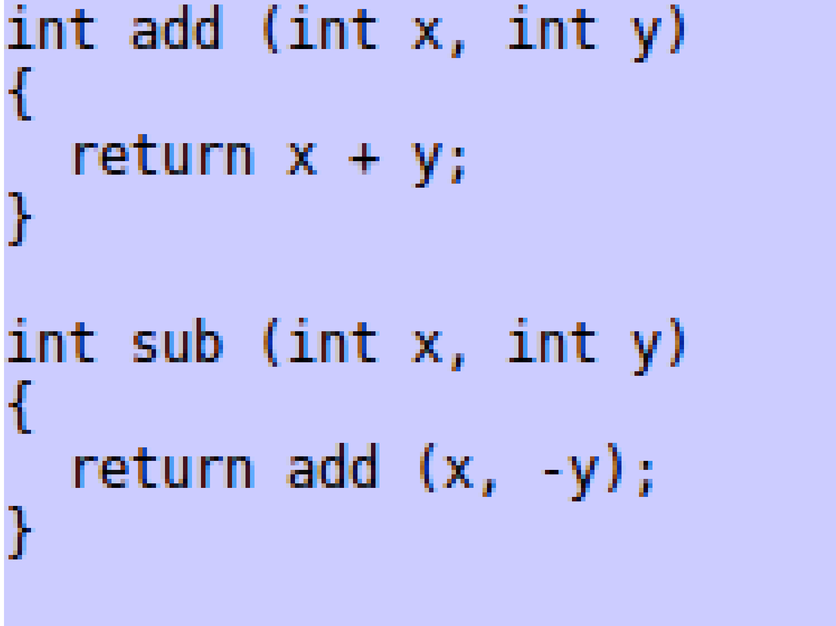
- The overhead associated with calling and returning from a function can be eliminated by:

◆ Expanding the body of the function inline,

◆ and then additional opportunities for optimization may be exposed as well.

Inlining

- In the code fragment below, the function add() can be expanded inline at the call site in the function sub().

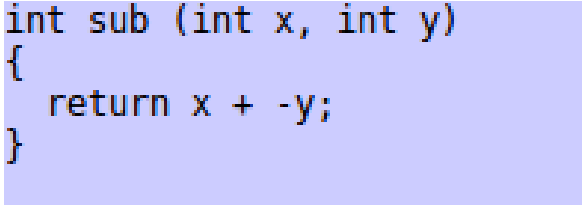


```
int add (int x, int y)
{
  return x + y;
}

int sub (int x, int y)
{
  return add (x, -y);
}
```

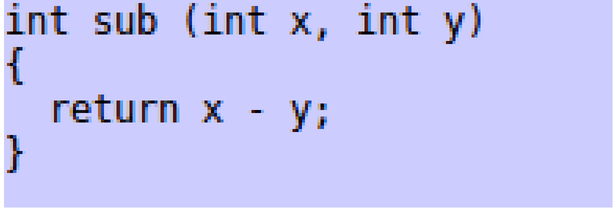
Inlining

- Expanding add() at the call site in sub() yields:



```
int sub (int x, int y)
{
  return x + -y;
}
```

- which can be further optimized to:



```
int sub (int x, int y)
{
  return x - y;
}
```

Control-Flow Simplification

- High-level optimization
- Idea:
  - If we know the value of a branch condition, eliminate the unused branch

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

- 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

Anatomy of an Optimization

- Two big parts:

- Program analysis - *Pass over code to find:*
  - Opportunities
  - Satisfy safety constraints

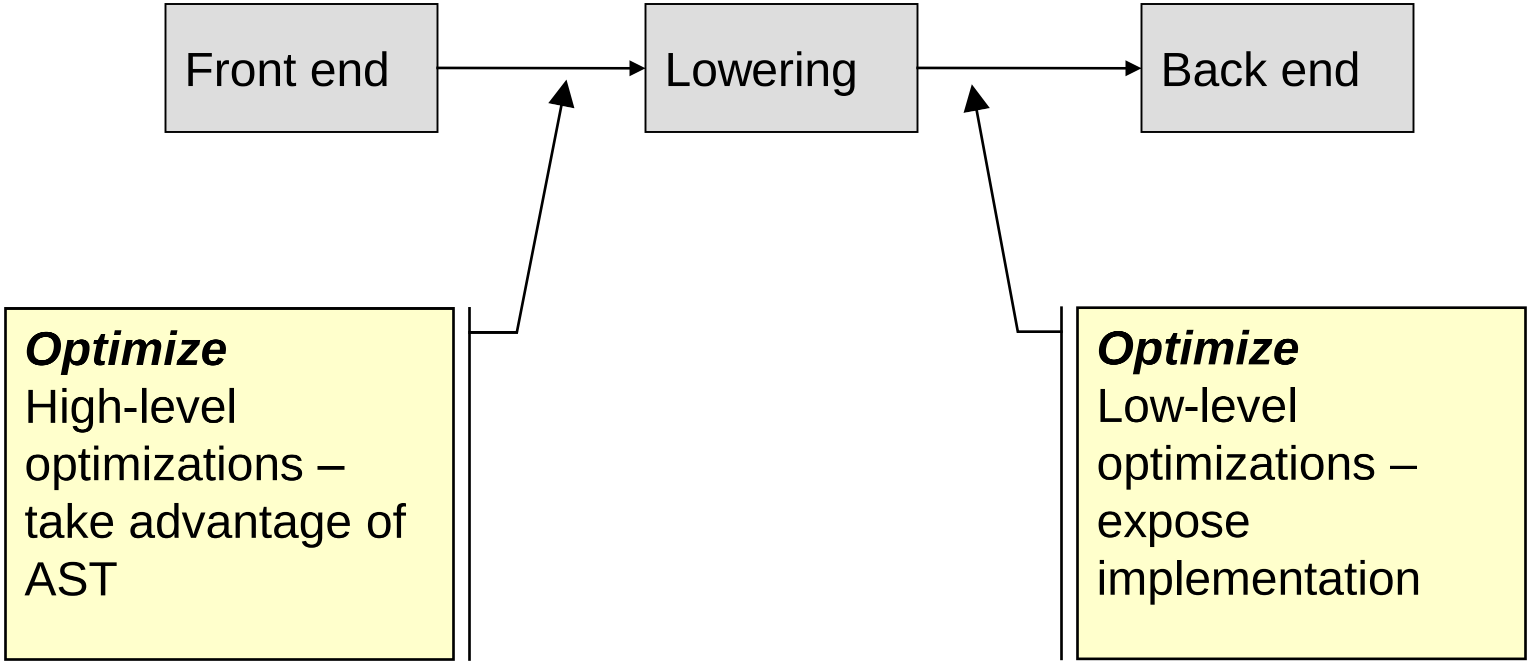
- Program transformation
  - Change the code to exploit opportunity

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



Front end

Lowering

Back end

Optimize
High-level optimizations – take advantage of AST

Optimize
Low-level optimizations – expose implementation

Writing Fast Programs

- In practice:
- Pick the right algorithms and data structures
  - Asymptotic complexity (Big O)
  - 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”

Optimizations

High-level IR

- Inlining
- Constant folding
- Algebraic simplification
- Constant propagation
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Branch prediction/optimization
- Register allocation
- Loop unrolling
- Cache optimization

Low-level IR

Scope of Optimization

- Local (or single block)
  - Confined to straight-line code
  - Simplest to analyze

- Intraprocedural (or global)
  - Consider the whole procedure

- Interprocedural (or whole program)
  - Consider the whole program

Summary

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

Where are We

- We have;
  - recognize tokens
  - Accept true statements
  - Verify meaning to statements
  - Put these statements in a neutral format
  - Optimize time and memory for code
- We have not;
  - Matched IR to specific assembly language
  - Allocated IR to memory and register

Where are We

- As first course in compilers, in 45hrs, we have achieved allot
- As a student of language theory and compiler design, the appetite has just been created
- Go out there, settle to;
  - Under stand more theory
  - Realize the theories from easiest to the furthest you can reach

Thank You