Code Optimization

1

Code Optimization

- Code generation techniques and transformations that result in a semantically equivalent program that runs more efficiently
  - faster
  - uses less memory
  - or both
- Often involves a time-space tradeoff. Techniques that make the code faster often require additional memory, and conversely
- Term "optimization" is actually used improperly
  - generated code is rarely optimal
  - better name might be "code improvements"

2

Code Optimization
(continued)

- Optimizing compilers
- May be performed on intermediate representations of the program
  - high level representation such as abstract syntax trees
  - machine code or a low-level representation
- Local versus global optimizations (DEC Ada PL/I story)
- Machine-dependent versus machine-independent optimizations

> "There is no such thing as a machine-independent optimization." – William A. Wulf

3

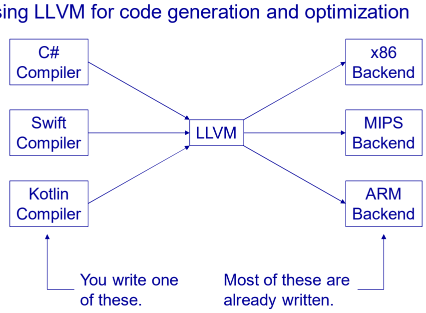
Guidelines for Optimization

- Make it correct before making it faster.
- The best source of optimization is often the programmer.
  - better algorithm (bubble sort versus quick sort)
  - profiling to determine areas where optimization matters
  - rewriting time-critical code in assembly language
- Test compiler both with and without optimizations.
- Let someone else do it.
  - e.g., use a common, low-level intermediate language (LLVM)
- Remember that occasionally, especially during development, faster compile times can be more important that more efficient object code.

4

LLVM

- Using LLVM for code generation and optimization



5

Code Optimization Issues

- Often difficult to improve algorithmic complexity
- Compilers must support a variety of conflicting objectives
  - cost of implementation
  - schedule for implementation
  - compilation speed
  - runtime performance
  - size of object code
- Overhead of compiler optimization
  - extra work takes time
  - whole-program optimization is time consuming and often difficult or impractical

6

Common Optimization Themes

- Optimize the common case
  - even at the expense of a slow path
- Less code
  - usually results in faster execution
  - lower product cost for embedded systems
- Exploit the memory hierarchy
  - registers first, then cache, then main memory, then disk
- Parallelize
  - allow multiple computations to happen in parallel
- Improve Locality
  - related code and data placed close together in memory

©SoftMoore Consulting Slide 7

7

Optimization: Machine-Specific Instructions

- Use of specific instructions available on the target computer
- Examples
  - increment and decrement instructions in place of add instructions
  - block move instructions
  - array-addressing instructions
  - pre/post increment instructions

©SoftMoore Consulting Slide 8

8

Optimization: Register Allocation

- Efficient use of registers to hold operands
- Register allocation – selection of variables that will reside in registers (e.g., a loop index)
- Register assignment – selection of specific registers for the variables
- Very hard problem – one common approach uses a "graph coloring" algorithm.

©SoftMoore Consulting Slide 9

9

Optimization: Constant Folding

- Compile-time evaluation of arithmetic expressions involving constants
- Example: Consider the assignment statement
  `c = 2*PI*r;`
  Assuming `PI` has been declared as a named constant, evaluation of `2*PI` can be performed by the compiler rather computed at runtime, and the resulting product can be used in the expression.

©SoftMoore Consulting Slide 10

10

Optimization: Algebraic Identities

- Use of algebraic identities to simplify certain expressions
- Examples

```
x + 0 = 0 + x = x
x*1 = 1*x = x
0/x = 0 (provided x ≠ 0)
x - 0 = x
0 – x = -x
```

©SoftMoore Consulting Slide 11

11

Optimization: Strength Reduction

- Replacing operations with simpler, more efficient operations
- Use of machine-specific instructions can be considered a form of strength reduction.
- Examples
  `i = i + 1` → `inc i` (use increment instruction)
  `i*2 or 2*i` → `i + i` (replace multiplication by 2 with addition)
  `x/8` → `x >> 3` (replace division by 2^n with right-shift n)
  `MOV EAX, 0` → `XOR EAX` (smaller and faster)

©SoftMoore Consulting Slide 12

12

Optimization: Common Subexpression Elimination

- Detecting a common subexpression, evaluating it only once, and then referencing the common value
- Example: Consider the two following sets of statements

```
a = x + y;              a = x + y;
…                       …
b = (x + y)/2;          b = a/2;
```

- These two sets of statement are equivalent provided that x and y do not change values in the intermediate statements.

 Slide 13

13

Optimization: Loop-Invariant Code Motion
(a.k.a. Code Hoisting)

- Move calculations outside of a loop (usually before the loop) without affecting the semantics of the program.
  – also facilitates storing constant values in registers
- Example (from Wikipedia)

```
while j < maximum - 1 loop
    j = j + (4+a[k])*PI+5;   // a is an array
end loop;
```

The calculation of "`maximum - 1`" and "`(4+a[k])*PI+5`" can be moved outside the loop and precalculated.

```
int maxval  = maximum - 1;
int calcval = (4+a[k])*PI+5;
while j < maxval loop
    j = j + calcval;
end loop;
```

 Slide 14

14

Dead Code Elimination

- Dead (a.k.a., unreachable) code is code that can never be executed.
- Example. The following Java code adds debugging feedback to a program based on the value of a boolean variable.

```
private static final boolean DEBUG = false;
...

if (DEBUG)
  {
    ...
  }
```

 Slide 15

15

Dead Code Elimination
(continued)

- Since `DEBUG` is final, both the declaration of `DEBUG` and the entire `if` statement can be removed without affecting the program results.
- Like many other compilers, the Java compiler will perform this optimization.
- An analogous example in Kotlin is similarly optimized by the Kotlin compiler.
- Note that dead code elimination affects only the size of the generated code, not the speed at which it executes.

 Slide 16

16

Peephole Optimization

- Applied to the generated target machine code or a low-level intermediate representation.
- Basic idea: Analyze a small sequence of instructions at a time (the peephole) for possible performance improvements.
- The peephole is a small window into the generated code.
- Examples of peephole optimizations
  – elimination of redundant loads and stores
  – elimination of branch instructions to other branch instructions
  – algebraic identities and strength reduction
    (can be easier to detect in the target machine code)

 Slide 17

17

Example: Peephole Optimization

Source Code

```
...
loop
   ...

   exit when x > 0;
end loop;
...
```

Target Code

```
L4:
   ...
   LDLADDR 0
   LOADW
   LDCINT 0
   CMP
   BG L5
   BR L4
L5:
   BR L9
L8:
   LDLADDR 0
   LOADW
   LDCINT 0
   ...
```

peephole

Optimization: Replace

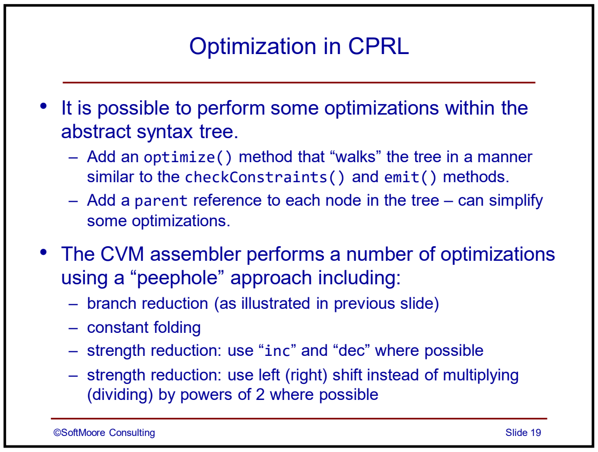
```
BG L5
BR L4
```

with

```
BLE L4
```

 Slide 18

18


Optimization in CPRL
• It is possible to perform some optimizations within the abstract syntax tree.
– Add an optimize() method that "walks" the tree in a manner similar to the checkConstraints() and emit() methods.
– Add a parent reference to each node in the tree – can simplify some optimizations.
• The CVM assembler performs a number of optimizations using a "peephole" approach including:
– branch reduction (as illustrated in previous slide)
– constant folding
– strength reduction: use "inc" and "dec" where possible
– strength reduction: use left (right) shift instead of multiplying (dividing) by powers of 2 where possible
©SoftMoore Consulting
Slide 19

19