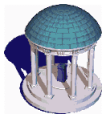


The University of North Carolina at Chapel Hill

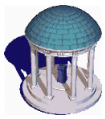
COMP 144 Programming Language Concepts

Spring 2002

Lecture 34: Code Optimization

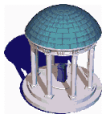
Felix Hernandez-Campos

April 17



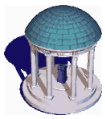
Optimization

- We will discuss code optimization from the point of view of the compiler and the programmer
- Code improvement is an important phase in production compilers
  - Generate code that *runs fast* (CPU-oriented)
    - » In some case, optimize for memory requirements or network performance
- Optimization is an important task when developing resource-intensive application
  - *E.g.* Games, Large-scale services



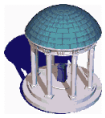
Optimization

- Two lectures on optimization
- References
  - **Scott's Chapter 13**
  - Doug Bell, *Make Java fast: Optimize!*
    - » http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize_p.html
  - *HyperProf*- Java profile browser
    - » http://www.physics.orst.edu/~bulatov/HyperProf/
  - *Python Performance Tips*
    - » http://manatee.mojam.com/~skip/python/fastpython.html
  - *Python Patterns - An Optimization Anecdote*
    - » http://www.python.org/doc/essays/list2str.html



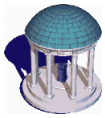
The Golden Rules of Optimization
Premature Optimization is Evil

- Donald Knuth, ***premature optimization is the root of all evil***
  - Optimization can introduce new, subtle bugs
  - Optimization usually makes code harder to understand and maintain
- Get your code right first, then, if really needed, optimize it
  - Document optimizations carefully
  - Keep the non-optimized version handy, or even as a comment in your code



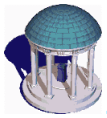
The Golden Rules of Optimization
The 80/20 Rule

- In general, ***80% percent of a program's execution time is spent executing 20% of the code***
- 90%/10% for performance-hungry programs
- Spend your time optimizing the important 10/20% of your program
- Optimize the common case even at the cost of making the uncommon case slower

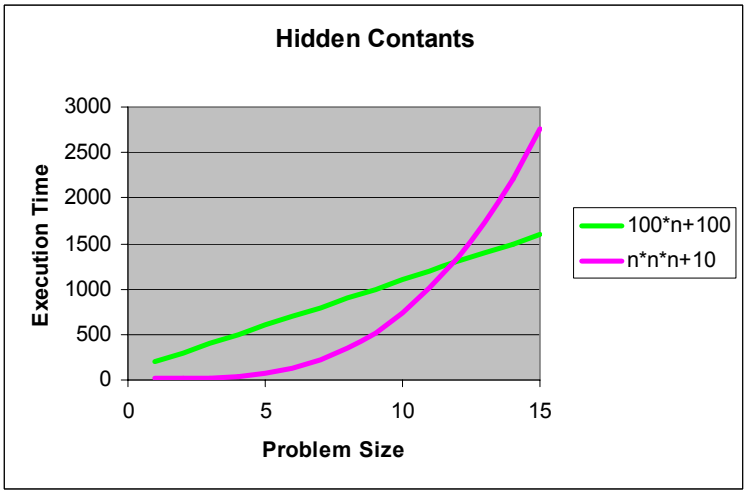


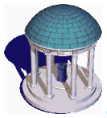
The Golden Rules of Optimization
Good Algorithms Rule

- The best and most important way of optimizing a program is using **good algorithms**
  - E.g. O(n*log) rather than O(n^2)
- However, we still need lower level optimization to get more of our programs
- In addition, asymthotic complexity is not always an appropriate metric of efficiency
  - Hidden constant may be misleading
  - *E.g.* a linear time algorithm than runs in 100**n*+100 time is slower than a cubic time algorithm than runs in n^3+10 time if the problem size is small



Asymptotic Complexity
Hidden Constants





General Optimization Techniques

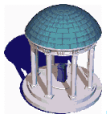
- **Strength reduction**
  - Use the fastest version of an operation
  - *E.g.*

```
x >> 2        instead of      x / 4
x << 1        instead of      x * 2
```

- **Common sub expression elimination**
  - Eliminate redundant calculations
  - *E.g.*

```
double x = d * (lim / max) * sx;
double y = d * (lim / max) * sy;

double depth = d * (lim / max);
double x = depth * sx;
double y = depth * sy;
```

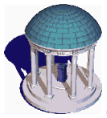


General Optimization Techniques

- **Code motion**
  - *Invariant* expressions should be executed only once
  - *E.g.*

```
for (int i = 0; i < x.length; i++)
    x[i] *= Math.PI * Math.cos(y);
```

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;
```

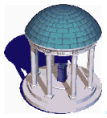
General Optimization Techniques

- **Loop unrolling**
  - The overhead of the loop control code can be reduced by executing more than one iteration in the body of the loop
  - *E.g.*

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;

double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i += 2) {
    x[i] *= picosy;
    x[i+1] *= picosy;
}
```

A efficient "+1" in array indexing is required

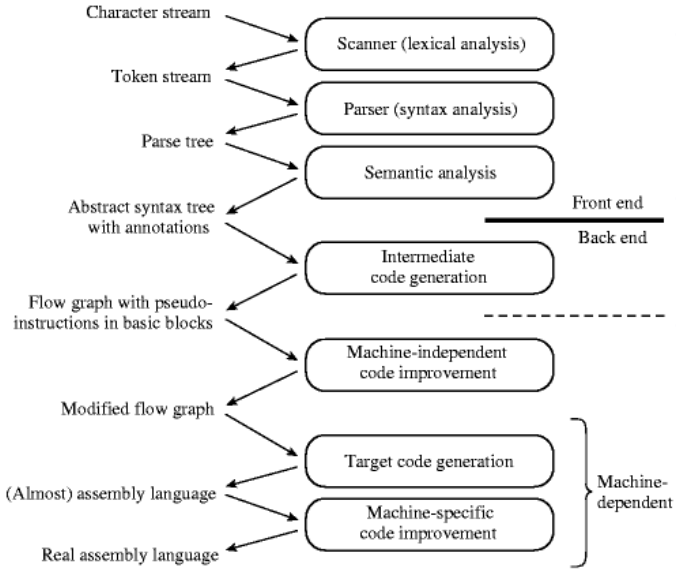


Compiler Optimizations

- Compilers try to generate *good* code
  - *I.e.* Fast
- Code improvement is challenging
  - Many problems are NP-hard
- Code improvement may slow down the compilation process
  - In some domains, such as just-in-time compilation, compilation speed is critical

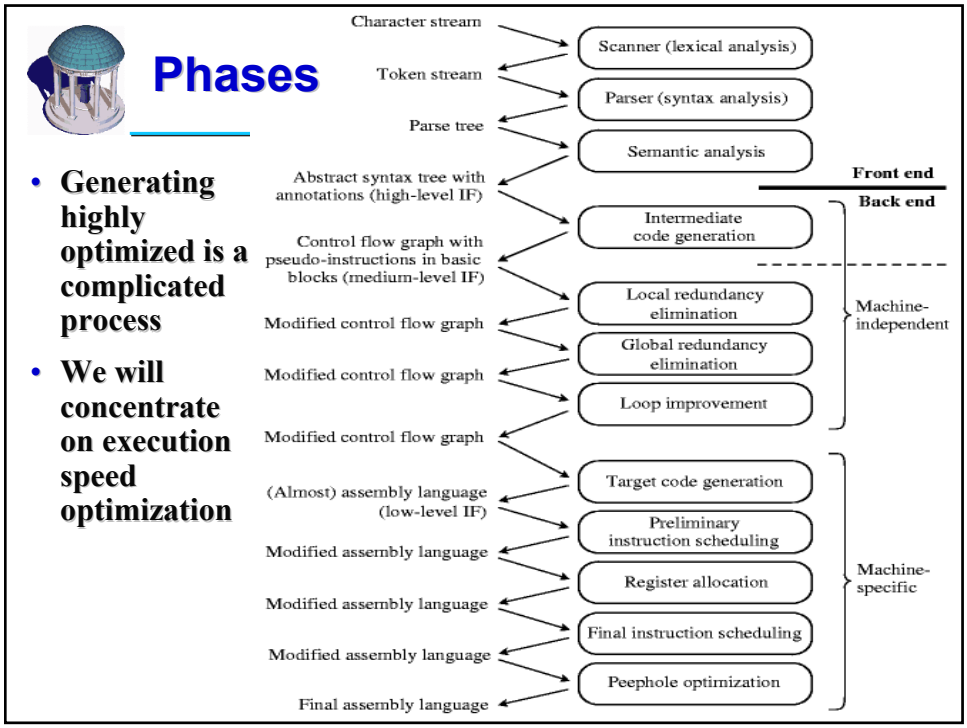
COMP 144 Programming Language Concepts
Felix Hernandez-Campos
11

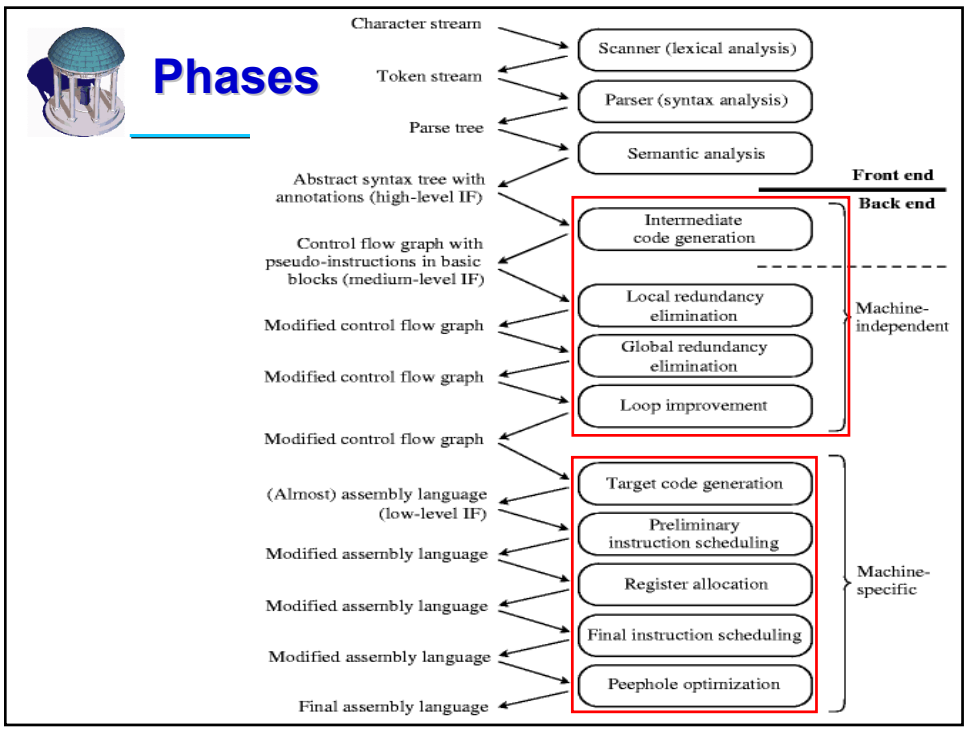
Phases of Compilation

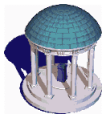


- The first three phases are *language-dependent*
- The last two are *machine-dependent*
- The middle two dependent on neither the language nor the machine

12

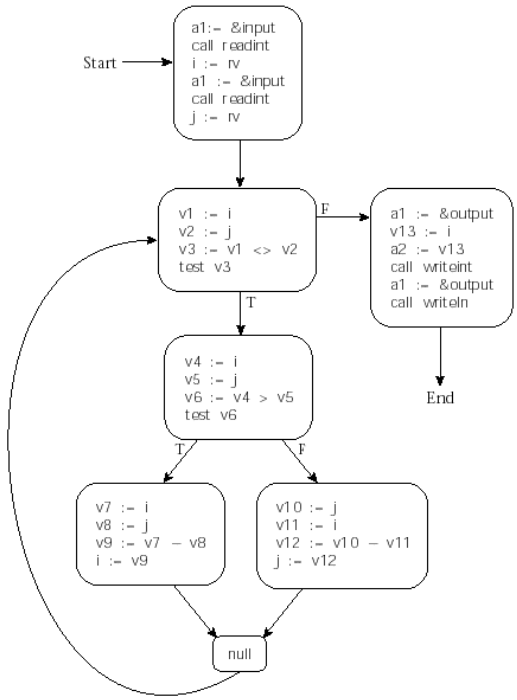

Phases
Generating highly optimized is a complicated process
We will concentrate on execution speed optimization
Character stream
Token stream
Parse tree
Abstract syntax tree with annotations (high-level IF)
Control flow graph with pseudo-instructions in basic blocks (medium-level IF)
Modified control flow graph
Modified control flow graph
Modified control flow graph
(Almost) assembly language (low-level IF)
Modified assembly language
Modified assembly language
Modified assembly language
Final assembly language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis
Front end
Back end
Intermediate code generation
Local redundancy elimination
Global redundancy elimination
Loop improvement
Machine-independent
Target code generation
Preliminary instruction scheduling
Register allocation
Final instruction scheduling
Peephole optimization
Machine-specific


Phases
Character stream
Token stream
Parse tree
Abstract syntax tree with annotations (high-level IF)
Control flow graph with pseudo-instructions in basic blocks (medium-level IF)
Modified control flow graph
Modified control flow graph
Modified control flow graph
(Almost) assembly language (low-level IF)
Modified assembly language
Modified assembly language
Modified assembly language
Final assembly language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis
Front end
Back end
Intermediate code generation
Local redundancy elimination
Global redundancy elimination
Loop improvement
Machine-independent
Target code generation
Preliminary instruction scheduling
Register allocation
Final instruction scheduling
Peephole optimization
Machine-specific

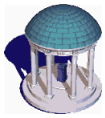


Example
Control Flow Graph

- ***Basic blocks*** are maximal-length set of sequential operations
  - Operations on a set of *virtual registers*
    - » Unlimited
    - » A new one for each computed value
- Arcs represent interblock control flow



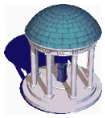
COMP 144 I
Fe



Peephole Optimization

- Simple compiler do not perform machine-independent code improvement
  - They generates *naïve* code
- It is possible to take the target hole and optimize it
  - Sub-optimal sequences of instructions that match an optimization pattern are transformed into optimal sequences of instructions
  - This technique is known as ***peephole optimization***
  - Peephole optimization usually works by sliding a window of several instructions (a *peephole*)

COMP 144 Programming Language Concepts
Felix Hernandez-Campos
16



Peephole Optimization
Common Techniques

Elimination of redundant loads and stores

```
r2 := r1 + 5
i := r2
r3 := i
r4 := r3 × 3
```

becomes

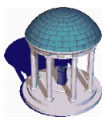
```
r2 := r1 + 5
i := r2
r4 := r2 × 3
```

Constant folding

```
r2 := 3 × 2
```

becomes

```
r2 := 6
```

Peephole Optimization
Common Techniques

Constant propagation

```
r2 := 4
r3 := r1 + r2
r2 := ...
```

becomes

```
r2 := 4
r3 := r1 + 4
r2 := ...
```

and then

```
r3 := r1 + 4
r2 := ...
```

```
r2 := 4
r3 := r1 + r2
r3 := *r3
```

becomes

```
r3 := r1 + 4
r3 := *r3
```

and then

```
r3 := *(r1+4)
```

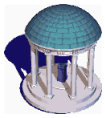
```
r1 := 3
r2 := r1 × 2
```

becomes

```
r1 := 3
r2 := 3 × 2
```

and then

```
r1 := 3
r2 := 6
```



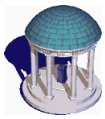
Peephole Optimization
Common Techniques

Copy propagation

```
r2 := r1                 r2 := r1
r3 := r1 + r2   becomes  r3 := r1 + r1   and then   r3 := r1 + r1
r2 := 5                  r2 := 5                    r2 := 5
```

Strength reduction

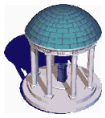
```
r1 := r2 × 2   becomes   r1 := r2 + r2   or   r1 := r2 << 1
r1 := r2 / 2   becomes   r1 := r2 >> 1
r1 := r2 × 0   becomes   r1 := 0
```

Peephole Optimization
Common Techniques

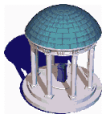
Elimination of useless instructions

```
r1 := r1 + 0
r1 := r1 × 1
```



Peephole Optimization

- Peephole optimization is very fast
  - Small overhead per instruction since they use a small, fixed-size window
- It is often easier to generate naïve code and run peephole optimization than generating good code!



Reading Assignment

- Read Scott
  - Ch. 13 intro
  - Sect. 13.1
  - Sect. 13.2
- Doug Bell, *Make Java fast: Optimize!*
  - http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize_p.html