Code Analysis and Optimization

Pat Morin
COMP 3002
Outline

• Basic blocks and flow graphs
• Local register allocation
• Global register allocation
• Selected optimization topics
The Big Picture

- By now, we know enough to compile a programming language into machine code
- But the machine code isn't terribly efficient
Today's Lecture

• We will look at different kinds of optimizations a compiler can perform
• Different optimizations apply to different architectures or at different times
  – Virtual stack machines
  – 3-Address instructions
  – Register-based machines
**Basic Blocks**

- A basic block is a block of (machine or intermediate) code that always runs straight through without interruption.
- A *block head* is
  - the target of a (conditional or unconditional) jump, or
  - the code immediately after a jump or function call, or
  - the first line of code in a function.
- A *basic block* starts at a block head and continues to the next block head (or the end of the code/function).
Basic Block Example

```
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 0
ifeq false_label

ldc "true"
goto print_it

false_label:
  ldc "false"

print_it:
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

return
```
Basic Block Example

- Identify the basic blocks in the following

```java
ldc 0.0
fstore 1

start:
  fload 1 ; load i  
  fload 0 ; load n  
  fcmpl
  ifge done
  fload 1
  invokestatic SimpleTest/printFloat(F)V
  fload 1
  ldc 1.0
  fadd
  fstore 1
  goto start

done:  
  return
```
Why Basic Blocks?

• Because basic blocks always run straight through, without interruption
  – We are free to modify a lot of the code within a basic block
  – If a variable is set within a basic block then we know the value of that variable for the remainder of the block
Transformations on Basic Blocks

• Common subexpression elimination
  – Works because we know the values of all variables that have been set within that block

\[
\begin{align*}
a & := b + c \\
b & := a - d \\
c & := b + c \\
d & := a - d ; \text{ replace with } d := b
\end{align*}
\]
Transformations on Basic Blocks

• Useless code elimination
  – We can determine that some statements have no effect outside the basic block and can be eliminated

```
iload 0
ldc 1
iadd
istore 0
iload 0 ; eliminate
pop ; eliminate
```
Transformation on Basic Blocks

- Renaming temporary variables (3-address codes) and reordering instructions can be useful

\[
t_1 := b+c \\
t_2 := x+y \quad ; \text{can reorder if } b,c\neq t_2 \text{ and } x,y\neq t_1
\]
Transformations on Basic Blocks

- We can use algebraic identities to simplify code or use less expensive instructions
  - Usually applies when one of the operands is a constant

\[
\begin{align*}
x &:= x + 0 \quad ; \text{eliminate} \\
x &:= x \times 1 \quad ; \text{eliminate} \\
x &:= y + 0 \quad ; x := y \\
x &:= y \times 1 \quad ; x := y \\
x &:= y \times 2 \quad ; x := y + y \quad \text{might be faster}
\end{align*}
\]
Register Machines
Register Machines

- A typical computer has a fixed number of registers
- All operations require that the operands be contained in these registers
- Reading data from memory into registers (load) and writing it back (store) is slow
- We want to minimize the number of loads and stores
- Problem: Many functions will have more variables than available registers
Next-Use Information

- When inspecting a basic block, it can be helpful to know when each variable will be used next.

```
; code for x := y + z
mov y, R0     ; put y into register 0
mov z, R1     ; put z into register 1
add R0, R1    ; store result of add in R0
mov R0, x     ; store x

; code for p := y * 2
mov y, R0     ; put y into register 0
ld  2, R1     ; put 2 into register 1
add R0, R1    ; store result of add in R0
mov R0, p     ; store p
```
Next-Use Information (Cont'd)

• An improved use of registers

```assembly
; code for x := y + z
mov y, R0     ; put y into register 0
mov z, R1     ; put z into register 1
add R1, R0    ; store result of add in R1
mov R1, x     ; store x

; code for p := y * 2
; y is still in R0
ld  2, R1     ; put 2 into register 1
add R0, R1    ; store result of add in R0
mov R0, p     ; store p
```
Computing Next Use Information

• By scanning backwards we can compute next-use information for each variable used in each line of a basic block

• With each variable, we know
  – the next time it is used in an expression
  – the next time its value is changed

• Aliasing (pointers and references) can complicate matters
Next-Use Information - Example

1. t1 := b * b ; t1(5) b(never)
2. t2 := 4 * a ; t2(3) a(6)
3. t3 := t2 * c ; t3(4) t2(never) c(never)
4. t4 := sqrt(t3) ; t4(5) t3(never)
5. t5 := t1 – t4 ; t5(7) t1(never) t4(never)
6. t6 := 2 * a ; t6(6) a(never)
7. t7 := t5 / t6 ; t7(8) t5(never) t6(never)
Generating Code From Next-Use

- Scan the block from beginning to end, keeping track of where each variable is stored (in which register or in memory)

- To generate code for \( x := y + z \)
  - Assume \( x, y, \) and \( z \) are distinct
  - If \( x \) is in a register \( R_i \) then mark \( R_i \) as free
  - If \( y \) and \( z \) are not in registers, then bring them into registers
  - Do the addition (now \( x \) is stored in a register)
**Bringing a Variable into a Register**

- To load a variable $y$ into a register
  - If some register is free then use that register
  - Otherwise, consider registers that store values also stored in memory and use one of those
  - Otherwise, write a register into memory and use it

- In the case of ties, write the register holding the variable whose next use information is farthest into the future

- At the end of the basic block, generate code to write all registers back to memory
Code Generation - Example

- Generate code for this on a 2-register machine

1. \( t_1 := b \times b \); \( t_1(5) \) b(never)
2. \( t_2 := 4 \times a \); \( t_2(3) \) a(6)
3. \( t_3 := t_2 \times c \); \( t_3(4) \) t_2(never) c(never)
4. \( t_4 := \sqrt{t_3} \); \( t_4(5) \) t_3(never)
5. \( t_5 := t_1 - t_4 \); \( t_5(7) \) t_1(never) t_4(never)
6. \( t_6 := 2 \times a \); \( t_6(6) \) a(never)
7. \( t_7 := t_5 / t_6 \); \( t_7(8) \) t_5(never) t_6(never)
The Pains of Pointers

• In languages with pointers, basic register allocation becomes much more difficult
  – This is especially true in languages, like C and C++ with very flexible pointers

• For this reason, many languages outperform even the best optimizing C compilers

```c
int *a;
int x, y, z, w;

... *a = 23; // this may have modified x, y, z, or w
    // a C compiler has to work hard to
    // know that it doesn't
```
The Control Flow Graph
The Control Flow Graph

• The (control) flow graph is a directed graph whose vertices are the basic blocks.

• An edge goes from block A to block B if
  – A terminates with a (conditional) jump to B, or
  – B comes after A and A's last statement is anything other than a goto or return (unconditional jump).

• The flow graph tells us, for every block, which blocks we might visit next.
getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 0
    ifeq false_label

ldc "true"
    goto print_it

false_label:
    ldc "false"

print_it:
    invokevirtual
    java/io/PrintStream/println(Ljava/lang/String;)V

return
Flow Graph Example

- Construct the control flow graph:

```java
ldc 0.0
fstore 1

start:
  fload 1     ; load i
  fload 0     ; load n
  fcmpl
  ifge done
  fload 1
  invokestatic SimpleTest/printFloat(F)V
  fload 1
  ldc 1.0
  fadd
  fstore 1
  goto start

done:
  return
```
Global Register Allocation

• We have seen an efficient algorithm for managing registers within a block
  – Summary:
    • Keep track of which values are in which registers
    • Only store a register when necessary
    • Store all “dirty” registers at the end of a block

• Problem:
  – It's often worth keeping registers in variables across blocks
    • loop indices are a common example
Example

```
i := 0
start: i := i + 1
    ...
    if i < 1000 goto start
```

```
ldc R0, 0
start: inc R0
    ...
    ldc R1, 1000
    sub R1, R0
    jmplt R1, start
```

```
ldc R0, 0
mov R0, o  ; store i
start: mov i, R0  ; load i
    inc R0
    ...
    ldc R1, 1000
    sub R1, R0
    mov R0, i  ; store i
    jmplt R1, start
```
Global Register Allocation

• Designate one or more registers as “variable registers” that will be used to store local variables
• Analyze loops and decide which variables get to become “register” variables
Assigning “Register” Variables

- Easy case: 1 block in a loop
  - Calculate the savings for each variable
    - save 1 load if the variable is accessed
    - save 1 store if the variable is modified

- Example:
  - i used and modified (1 load + 1 store)
  - a is modified but not used (1 store)
  - b and c are used but not modified (1 load)
  - putting i in a register yields the greatest savings

```python
start: i := i + 1
a := b + c
...
if i < 1000 goto start
```
More Complicated Variants

• A cycle with an if statement
  – Only count savings by half as much in the red boxes
More Complicated Variants

• Nested Cycles
  – Pay a penalty for choosing a different variable to use in the inner cycle
Other Control Flow Graph Tricks

• The control flow graph allows several other useful optimizations based on reachability analysis

• Can we get to a basic block B from a basic block A?

• This question is answered by computing the transitive closure of the control flow graph
Dead Code Elimination

• A piece of code is *dead* if it cannot be reached in any execution path.

• For a function
  – look at the first basic block of the function (A)
  – code B is dead if A->B is not in the transitive closure.

• Dead code never executes and can therefore be eliminated.
No Longer Used Variables

- At some point during the execution of a function, a local variable may never be used again
  - We can avoid unnecessarily storing this variable
- If variable i is modified in basic block A
  - Check if there is any block B such that
    - i is used in block B, and
    - A -> B in the transitive closure
  - If not, then i is never used again after visiting A
When to Construct the Flow Graph

- The best time to construct the control flow graph is after some optimizations have been done on the basic blocks.
- This may reduce the number of edges in the graph.

```plaintext
start:
  ...
  t0 = 1 < 3
  if t0 goto start
```
Summary

• Basic blocks and control flow graphs represent a compiler's understanding of how a program executes.

• Basic blocks always run right through
  – We understand enough about values in basic blocks to optimize aggressively.

• Flow graphs represent execution paths
  – Give more information about data in basic blocks.
  – Allow for reachability analysis.