CSI2005
Principles of Compiler Design

MODULE - 6

Dr. WI. Sureshkumar
Associate Professor
School of Computer Science and Engineering (SCOPE)
VIT Vellore
wi.sureshkumar@vit.ac.in
SJT413A34

Code Optimization

- Basic Blocks and Flow Graphs
- The DAG Representation of Basic Blocks
- The Principal Sources of Optimization
- Optimization of Basic Blocks
- Loops in Flow Graphs
- Peephole Optimization
- Introduction to Global Data-Flow Analysis

Code Optimization

- 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 code
- Advantages
  - Faster execution
  - Efficient memory usage
  - Better performance

Code Optimization Techniques

- Compile time evaluation
  - Constant folding
  - Constant propagation
- Common sub-expression elimination
- Strength reduction
- Code motion
- Dead code elimination

Code Optimization Techniques

- Constant folding

  It refers to a technique if evaluating the expressions whose operands are known to be constant at compile time.

Ex. a = (22/ 7) * b;

- Constant propagation

  Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

Ex. In the code fragment below, the value of x can be propagated to the use of x.

```
x = 3;
y = x + 4;
```

Below is the code fragment after constant propagation and constant folding.

```
x = 3;
y = 7;
```

Code 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;
```

Code Optimization Techniques

- **Common sub expression elimination**

Before Optimization	After Optimization
t1 = 4 * i	t1 = 4 * i
t2 = a[t1]	t2 = a[t1]
t3 = 4 * j	t3 = 4 * j
t4 = 4 * i	t5 = n
t5 = n	t6 = b[t1] + t5
t6 = b[t4] + t5	

Code 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;
```

Code Optimization Techniques

- Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated.

Ex:

In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

```
int global;
void f ()
{
 int i; i = 1; /* dead store */
 global = 1; /* dead store */
 global = 2;
 return;
 global = 3; /* unreachable */
}
```

Below is the code fragment after dead code elimination.

```
int global;
void f ()
{
   global = 2;
   return;
}
```

Basic Blocks

- A **basic block** is a maximal sequence of consecutive three-address instructions with the following properties:
  - The flow of control can only enter the basic block through the 1st instruction.
  - Control will leave the block without halting or branching, except possibly at the last instruction.
- Basic blocks become the nodes of a **flow graph**, with edges indicating the order.

Examples

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j]=0.0

for i from 1 to 10 do
     a[i,i]=1.0
```

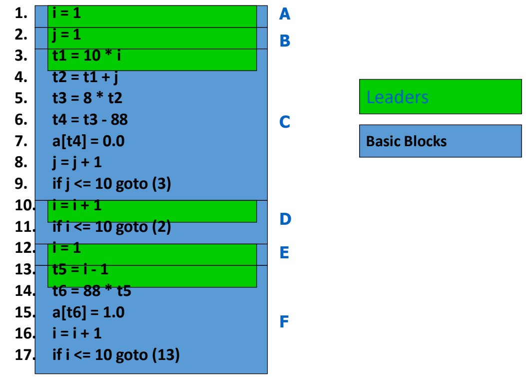
Identifying Basic Blocks

- Input: sequence of instructions *instr(i)*
- Output: A list of basic blocks
- Method:
  - Identify **leaders**:
    the first instruction of a basic block
  - Iterate: add subsequent instructions to basic block until we reach another leader

Identifying Leaders

- Rules for finding leaders in code
  - First instruction in the code is a leader
  - Any instruction that is the target of a (conditional or unconditional) jump is a leader
  - Any instruction that immediately follow a (conditional or unconditional) jump is a leader

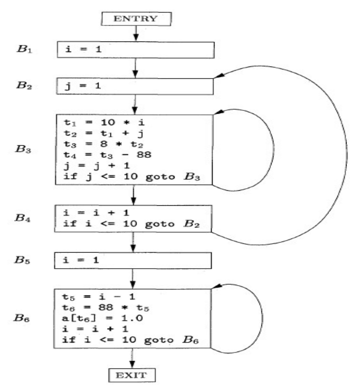
Basic Block Example



Control-Flow Graphs

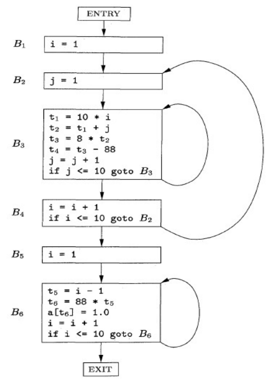
- **Control-flow graph**:
  - Node: an instruction or sequence of instructions (a basic block)
    - Two instructions i, j in same basic block *iff* execution of i *guarantees* execution of j
  - Directed edge: *potential* flow of control
  - Distinguished start node *Entry & Exit*
    - First & last instruction in program

CFG Example



Loop Examples

- {B3}
- {B6}
- {B2, B3, B4}



Peephole Optimization

- Introduction to peephole
- Common techniques
- Algebraic identities
- An example

Peephole Optimization

- Simple compiler do not perform machine-independent code improvement
  - They generates *naive* 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*)

Peephole Optimization

Goals:

- improve performance
- reduce memory footprint
- reduce code size

Method:

1. Exam short sequences of target instructions
2. Replacing the sequence by a more efficient one.

- redundant-instruction elimination
- algebraic simplifications
- flow-of-control optimizations
- use of machine idioms

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

Algebraic identities

- Worth recognizing single instructions with a constant operand
  - Eliminate computations
    - A * 1 = A
    - A * 0 = 0
    - A / 1 = A
  - Reduce strength
    - A * 2 = A + A
    - A/2 = A * 0.5
  - Constant folding
    - 2 * 3.14 = 6.28
- More delicate with floating-point

Replace Multiply by Shift

- A := A * 4;
  - Can be replaced by 2-bit left shift (signed/unsigned)
  - But must worry about overflow if language does
- A := A / 4;
  - If unsigned, can replace with shift right
  - But shift right arithmetic is a well-known problem
  - Language may allow it anyway (traditional C)

Addition chains for multiplication

- If multiply is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into sum of powers of two can be effective:
  - X * 125 = x * 128 - x*4 + x
  - two shifts, one subtract and one add, which may be faster than one multiply
  - Note similarity with efficient exponentiation method

Flow-of-control optimizations

```
    goto L1                  goto L2
     . . .                    . . .
L1: goto L2              L1: goto L2
```

```
    if a < b goto L1         if a < b goto L2
     . . .                    . . .
L1: goto L2              L1: goto L2
```

```
    goto L1                  if a < b goto L2
     . . .                   goto L3
L1: if a < b goto L2          . . .
L3:                      L3:
```

Peephole Opt: an Example

Source Code:

```
debug = 0
. . .
if(debug) {
   print debugging information
  }
```

Intermediate Code:

```
    debug = 0
    . . .
    if debug = 1 goto L1
    goto L2
L1: print debugging information
L2:
```

Eliminate Jump after Jump

Before:

```
    debug = 0
    . . .
    if debug = 1 goto L1
    goto L2
L1: print debugging information
L2:
```

After:

```
    debug = 0
    . . .
    if debug ≠ 1 goto L2
    print debugging information
L2:
```

Constant Propagation

Before:

```
    debug = 0
    . . .
    if debug ≠ 1 goto L2
    print debugging information
L2:
```

After:

```
    debug = 0
    . . .
    if 0 ≠ 1 goto L2
    print debugging information
L2:
```

Peephole Optimization Summary

- 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.

Loops in Flow Graphs

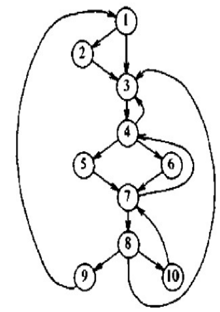
Dominators:

- A node d of a flow graph dominates node n (a node d is a dominator of a node n), written d dom n, if
  - Every path from the initial node of the flow graph to node n goes through d
  - Every node dominates itself
  - Entry node of a loop dominates all node in the loop.

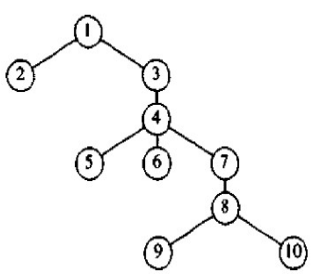
A useful way of presenting dominator information is in a tree, called a dominator tree.
- Initial node is a root
- Each node dominates only its descendants in the tree

Example:

1 dominates = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
2 dominates = { 2 }
3 dominates = { 3, 4, 5, 6, 7, 8, 9, 10}
4 dominates = { 4, 5, 6, 7, 8, 9, 10}
5 dominates = { 5 }
6 dominates = { 6}
7 dominates = { 7, 8, 9, 10}
8 dominates = { 8, 9, 10}
9 dominates = { 9 }
10 dominates = {10}



Dominator tree:



Natural Loops

One important application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of such loops:

1. A loop must have a single entry point, called the header. This entry point dominates all nodes in the loop.
2. There must be at least one way to iterate the loop i.e. at least one path back to the header.

A good way to find all the loops in a flow graph is to search for the edges whose heads dominate their tails.

If a→b is an edge, b is the head and a is the tail. We call such edges are back edges.

There is a edge 7→4, and 4 dom 7

There is a edge 4→3, and 3 dom 4

There is a edge 10→7, and 7 dom 10

There is a edge 8→3, and 3 dom 8

There is a edge 9→1, and 1 dom 9

- Given a back edge n→d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d.
- Node d is the header of the loop

The natural loop of the edge 10→7 consists of the nodes

7, 8 and 10

The natural loop of the edge 9→1 is the entire flow grah.

Example:

Consider the following program fragment:

```
Fact(x)
 {
  int f = 1;
  for(i=2; i<=x; i++)
    f = f * I;
  return(f);
 }
```

The three address –code representation of the program fragment is:

```
10 f = 1
11 i =2
12 If i <= x  goto 14
13  goto 18
14  f = f * i
15  t1 = i + 1
16  i = t1
17  goto 12
18  goto calling program
```

Leaders are 10

12, 14, 18

13

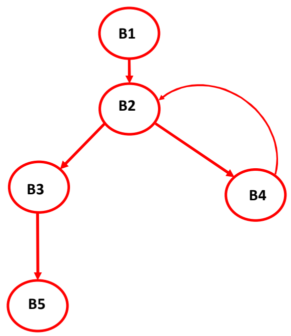
B1: 10, 11

B2: 12

B3: 13

B4: 14, 15, 16, 17

B5: 18



Find dominators and natural loops in the following flow graph: