Code Optimization

Code optimization:

- A transformation to a program to make it run faster and/or take up less space
- Code optimizations make improvements in codes by
  - Elimination of unnecessary operations
  - Replacing costly operations with some other operations
  - Predicting program behavior
  - Register allocation – machine dependent
- Optimization should be safe, preserve the meaning of a program.
- Code optimizations can be language dependent or language independent
  - e.g. stride-one access through array elements may be beneficial for some languages and not for some.

Code Optimization

- Code optimization can either be high level or low level:
  - High level code optimizations are machine independent
  - Example: Loop unrolling, loop fusion, procedure inlining
  - Low level code optimizations are machine dependent
  - Example: Instruction selection, register allocation
- Some optimization can be done in both levels:
  - Example: Common subexpression elimination, strength reduction, etc.
- Flow graph is a common intermediate representation for code optimization.

Basic Block and Control Flow Graph

- A basic block is a:
  - maximal-length sequence of instructions that will execute in its entirety
  - maximal-length straight-line code block
  - maximal-length code block with only one entry and one exit
- ... in the abscence of hardware faults, interrupts, crashes, threading problems, etc.
- To locate basic blocks in flattened code:
  - Starts with: (1) target of a branch (label) or (2) the instruction after a conditional branch
  - Ends with: (1) a branch or (2) the instruction before the target of a branch.
- A control flow graph is a graph whose nodes are basic blocks and whose edges are transitions between blocks.

Scope of Optimization

- Peephole optimization
  - Examine few instructions and transform into less expensive others
  - Peephole: a small moving window in the instruction sequence
  - Technique: examine a short sequence of target instructions (peephole) and try to replace it with a faster or shorter sequence
  - Example:
    - Redundant instruction elimination
    - Flow of control optimization
    - Algebraic simplifications
    - Instruction selection

Scope of Optimization

- Local optimization
  - Operate on a single basic block
  - Basic block: a sequence of consecutive statements with exactly 1 entry and 1 exit
- Loop optimization
  - Act on a number of basic blocks which made up a loop
- Global or intra-procedural optimizations
  - Act on the complete control flow graph of a single procedure
- Inter-procedural or whole program optimization
  - Most powerful, optimize interactions between procedures

Code Optimization

- Constant propagation
  - A transformation that propagates values defined in a statement of the form x=c, for a variable x and a constant c, through the entire program.
  - For the constant value, defined by a statement S_i, to be propagated the following conditions have to be satisfied:
    - There should be a statement S_j in which a use of the variable defined in S_i appears.
    - All definitions of the variable reaching the use at S_j must have the same value.
- Constant folding
  - Refers to the evaluation at compile time of expressions whose values are known to be constants. It involves determining that all operands in an expression are constants, performing the evaluation of the expression and replacing the expression by its value.
  - May not be always legal.
  - e.g. a) area = (22.0 / 7.0) * r ** 2

Code Optimization

- Useless code elimination
  - A statement is useless if it computes only values that are not used on any executable path leading from the statement
  - Often arises from other code transformations
  - (Dead code eliminations)
- Unreachable code elimination
  - Code that cannot be executed regardless of the input data.
  - Likely to arise as a result of other transformations
  -

Dead code elimination – example

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

After dead code elimination

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

Code Optimization

- Copy propagation
  - Given an assignment x=y for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y.
  - For a copy statement, S_l : x=y, we can replace any later use *u* of the variable x by the variable y if the following conditions are satisfied:
    - All definitions S_k of x reaching *u* must be a copy statement with variable y as the source operand (i.e. value of x does not change)
    - On every path from definition S_k to use *u*, including paths that go through *u* several times, there are no assignments to the source variable y (i.e. value of y also does not change after defining x)
  - An idea behind this technique is to use g for f whenever possible after the copy of f := g

Copy propagation

before
x := t3
a[t7] := t5
a[t10] := x
Goto b2

After
x := t3
a[t7] := t5
a[t10] := t3
Goto b2

Code Optimization

- Common sub-expression elimination
  - An occurrence of an expression in a program is a common subexpression if another occurrence of the expression exists whose evaluation precedes this one in execution order and if the operands of the expression remain unchanged between these two executions.
  - Removes re-computations of common subexpressions and replaces them with uses of saved values.
  - Identification of common subexpressions:
    - Search for lexically equivalent expressions, i.e. expressions consisting of the same identifiers, operators and appearing in the same order
    - The expressions must also be evaluated to identical values during any course of execution of the program (semantic equivalence)

```
R1 = M[R13+I] << 2
R1 = M[R1+_b]
R2 = M[R13+I] << 2;
R2 = M[R2+_b]
```

```
R1=M[R13+I] << 2
R1 = M[R1+_b]
R2 = R1
```

```
a = b * c
.....
x = b * c +5
```

```
Temp = b * c
a = temp
.....
x = temp + 5
```

Code Optimization

Code space reduction

```
if a < b then
   z = x ** 2
   ....
else
    y = x ** 2 + 19
```

```
temp = x ** 2
if a < b then
   z = temp
   ....
else
Y = temp + 19
```

- Strength reduction
  - Replace 2 * x with a shift operation or x^3 with x * x * x
  - OR

```
for k = 1 to 10 do
....
x = k * 5
....
end
```

```
temp = 5
for k = 1 to 10 do
....
x = temp
....
temp = temp + 5
end
```

Code Optimization

- Removing procedure calls
  - Procedure inlining – replace the procedure call with the code of the procedure body
  - Tail recursion removal (when the last operation is a call to itself)

```
int gcd (int u, int v)
{
   if (v==0) return u;
   else return gcd ( v, u % v );
}
```

```
int gcd (int u, int v)
{
   begin :
   if ( v == 0) return u;
   else
   {
      int t1 = v, t2 = u % v;
      u = t1; v = t2;
      goto begin;
   }
}
```

Code Optimization

- Loop invariant code motion
  - A computation in a loop is invariant if its value does not change within the loop.
  - Loop invariant code motion recognizes invariant computations in loops and moves them to the loop preheader.
  - For an invariant computation
  - s: x = y + z in a loop L, the following three conditions ensure that code motion does not change the semantics of the program
    - The basic block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with successor not in the loop
    - No other statement is in the loop that assigns to x
    - No use of x in the loop is reached by any definition of x from outside the loop.

```
e.g. while ( i < n )
do
     {  A = B + C;
        Z  = A * i;
        X = f (Z);
        i = i + 1;
      }
```

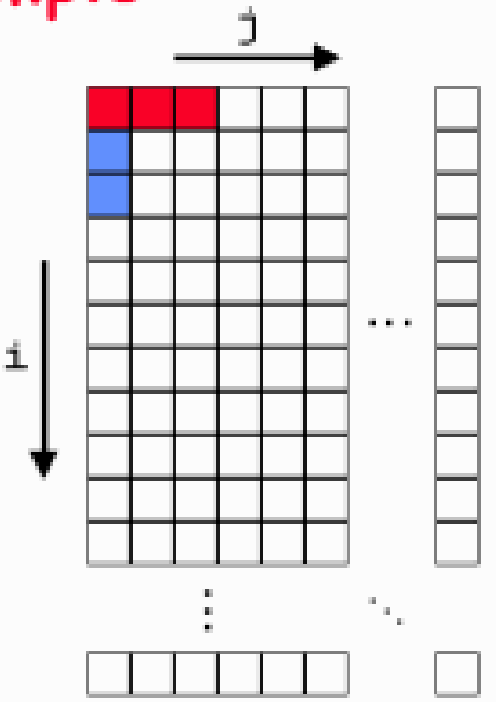
```
A = B + C
while ( i < n )
do
{  Z  = A * i;
   X = f (Z);
   i = i + 1;
 }
```

Code Optimization

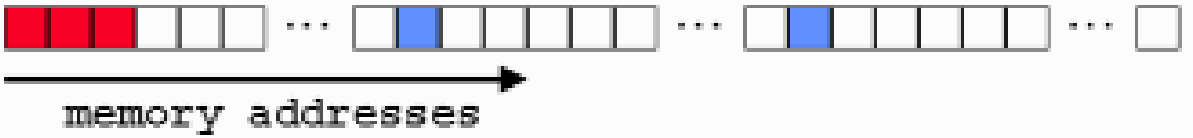
- Loop interchange
  - Exchanges the position of two loops in a tightly nested loop
  - Requirements for loop interchaging
    - The loops L1 and L2 must be tightly nested
    - The loop limits of L2 are invariant in L1
    - There are no statements Si and Sj in L2 with specific types of dependences
      - Not always legal
      - Improves locality of reference

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
          x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
          x[i][j] = 2 * x[i][j];
```



Sequential accesses instead of striding through memory every 100 words; improved spatial locality



Code Optimization

- Loop fusion
  - Fuses two adjacent loops into one
  - Reduces the loop overhead, and increases instruction parallelism
  - Requirements for fusing two loops L1 and L2 are
    - The two loop control variables should be the same and the loop limits must be identical
    - There should not be any statement S_v in L1 and S_w in L2 with some dependence relation
    - Both loops do not have a conditional branch that exits the loop
    - Loops L1 and L2, both should not have I/O statements
- Loop fission
  - Reverse of fusion
  - Improves locality of reference

Loop Fusion and Loop Fission

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

FUSION

```
for (i = 0; i < 300; i++)
 {
  a[i] = a[i] + 3;
  b[i] = b[i] + 4;
 }
```

```
for (i = 0; i < 300; i++)
 {
   a[i] = a[i] + 3;
   b[i] = b[i] + 4;
 }
```

FISSION

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

Code Optimization

- Loop reversal
  - Reversal changes the direction in which the loop traverses its iteration space
  - It is often used in conjunction with other loop transformations
  - Not always legal
  - e.g.

```
for (i=0; i<N; i++){
   a[i]=b[i]+1;
   c[i]=a[i]/2;
}
```

may be replaced with

```
for (i=N-1; i<=0; i--){
   a[i]=b[i]+1;
   c[i]=a[i]/2;
}
```

Code Optimization

- Loop unrolling
  - Decreases number of times the loop condition needs to be tested
  - Reduces number of jumps from end to beginning
  - Good for instruction pipelining and instruction level parallelism
  - However, for complete unrolling, number of instructions to be known at compile time
    - Increases code size significantly
- Loop unrolling and jam
  - Unroll and jam involves partially unrolling one or more loops
  - higher in the nest than the innermost loop, and fusing (“jamming”) the resulting loops back together.