Code Optimization
Overview and Examples

Code Optimization

- Why
  - Reduce programmers' burden
    - Allow programmers to concentrate on high level concept
    - Without worrying about performance issues
- Target
  - Reduce execution time
  - Reduce space
  - Sometimes, these are tradeoffs
- Types
  - Intermediate code level
    - We are looking at this part now
  - Machine code level
    - Instruction selection, register allocation, scheduling, cache opts etc

Code Optimization

- Scope
  - Peephole analysis
    - Within one or a few instructions
  - Local analysis
    - Within a basic block
  - Global analysis
    - Entire procedure or within a certain scope
  - Inter-procedural analysis
    - Beyond a procedure, consider the entire program

Code Optimization

- Techniques
  - Constant propagation
  - Constant folding
  - Algebraic simplification, strength reduction
  - Copy propagation
  - Common subexpression elimination
  - Unreachable code elimination
  - Dead code elimination
  - Loop Optimization
  - Function related
    - Function inlining, function cloning

Code Optimization Techniques

❖ Constant propagation

- If the value of a variable is a constant, then replace the variable by the constant
  - It is not the constant definition, but a variable is assigned to a constant
  - The variable may not always be a constant
- E.g.

  N := 10; C := 2;

  for (i:=0; i<N; i++) { s := s + i*C; }

  $\Rightarrow$ for (i:=0; i<10; i++) { s := s + i*2; }

  If (C) go to … $\Rightarrow$ go to …

  - The other branch, if any, can be eliminated by other optimizations
- Requirement:
  - After a constant assignment to the variable
  - Until next assignment of the variable
  - Perform data flow analysis to determine the propagation

Code Optimization Techniques

❖ Constant folding

- In a statement x := y op z or x := op y
- If y and z are constants
- Then the value can be computed at compilation time
- Example

  #define M 10

  x := 2 * M $\Rightarrow$ x := 20

  If (M < 0) goto L $\Rightarrow$ can be eliminated

  y := 10 * 5 $\Rightarrow$ y := 50
- Difference: constant propagation and folding
  - Propagation: only substitute a variable by its assigned constant
  - Folding: Consider variables whose values can be computed at compilation time and controls whose decision can be determined at compilation time

Code Optimization Techniques

- Algebraic simplification
  - More general form of constant folding, e.g.,
    - $x + 0 \Rightarrow x$ $\quad$ $x - 0 \Rightarrow x$
    - $x * 1 \Rightarrow x$ $\quad$ $x / 1 \Rightarrow x$
    - $x * 0 \Rightarrow 0$
  - Repeatedly apply the rules
    - $(y * 1 + 0) / 1 \Rightarrow y$
- Strength reduction
  - Replace expensive operations
    - E.g., $x := x * 8 \Rightarrow x := x << 3$

Code Optimization Techniques

- Copy propagation
  - Extension of constant propagation
  - After y is assigned to x, use y to replace x till x is assigned again
  - Example

    x := y; $\Rightarrow$ s := y * f(y)

    s := x * f(x)
  - Reduce the copying
  - If y is reassigned in between, then this action cannot be performed

Code Optimization Techniques

- Common subexpression elimination
  - Example:

a := b + c		a := b + c
c := b + c	⇒	c := a
d := b + c		d := b + c

  - Example in array index calculations
    - c[i+1] := a[i+1] + b[i+1]
    - During address computation, i+1 should be reused
    - Not visible in high level code, but in intermediate code
  - Applied using DAGs (Directed Acyclic Graph) as intermediate form

Code Optimization Techniques

- Unreacheable code elimination
  - Construct the control flow graph
  - Unreachable code block will not have an incoming edge
  - After constant propagation/folding, unreachable branches can be eliminated
- Dead code elimination
  - Ineffective statements
    - x := y + 1 (immediately redefined, eliminate!)
    - y := 5 ⇒ y := 5
    - x := 2 * z x := 2 * z
  - A variable is dead if it is never used after last definition
    - Eliminate assignments to dead variables
  - Need to do data flow analysis to find dead variables

Code Optimization Techniques

- Function inlining
  - Replace a function call with the body of the function
  - Save a lot of copying of the parameters, return address, etc.
- Function cloning
  - Create specialized code for a function for different calling parameters

Function inlining example

```
template<typename A, typename B>
inline auto Add(A a, B b) noexcept
{
return a + b;
}

int main()
{
int y = Add(5, 6);  ➔ replaced by int y = 5 + 6; ➔ y = 11;
}
```

Function cloning example

```
static int foo(int a, int b)
{
  if (b > 0)
    return a + b;
  else
    return a * b;
}
int bar(int m, int n)
{
  return foo(m, 5) + foo(m, n);
}
```

```
static int foo(int a, int b)
{
  if (b > 0)
    return a + b;
  else
    return a * b;
}
static int foo_clone(int a)
{
  return a + 5;
}
int bar(int m, int n)
{
  return foo_clone(m) + foo(m, n);
}
```

Code Optimization Techniques

- Loop optimization
  - Consumes 90% of the execution time

    $\Rightarrow$ a larger payoff to optimize the code within a loop
- Techniques
  - Loop invariant detection and code motion
  - Induction variable elimination
  - Strength reduction in loops
  - Loop unrolling
  - Loop peeling
  - Loop fusion

Code Optimization Techniques

- Loop invariant detection and code motion
  - If the result of a statement or expression does not change within a loop, and it has no external side-effect
  - Computation can be moved to outside of the loop
  - Example

    for (i=0; i<n; i++) a[i] := a[i] + x/y;

    - Three address code

      for (i=0; i<n; i++) { c := x/y; a[i] := a[i] + c; }

    $\Rightarrow$ c := x/y;

      for (i=0; i<n; i++) a[i] := a[i] + c;

Code Optimization Techniques

- Strength reduction in loops
  - Example

    s := 0; for (i=0; i<n; i++) { v := 4 * i; s := s + v;)

    ⇒ s := 0; for (i=0; i<n; i++) { v := v + 4; s := s + v;)
- Induction variable elimination
  - If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition
  - Example

    s := 0; for (i=0; i<n; i++) { s := 4 * i; … }   -- i is not referenced in loop

    ⇒ s := 0; e := 4*n; while (s < e) { s := s + 4; }

Code Optimization Techniques

- Loop unrolling
  - Execute loop body multiple times at each iteration
  - Get rid of the conditional branches, if possible
  - Allow optimization to cross multiple iterations of the loop
    - Especially for parallel instruction execution
  - Space time tradeoff
    - Increase in code size, reduce some instructions
- Loop peeling
  - Like unrolling
  - But unroll the first and/or last few iterations

Unrolling example

```
int countbit1(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

```
int countbit2(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        if (n & 2) bits++;
        if (n & 4) bits++;
        if (n & 8) bits++;
        n >>= 4;
    }
    return bits;
}
```

Code Optimization Techniques

❖ Loop fusion

❑ Example

```
for i=1 to N do
    A[i] = B[i] + 1
endfor
for i=1 to N do
    C[i] = A[i] / 2
endfor
for i=1 to N do
    D[i] = 1 / C[i+1]
endfor
```

Before Loop Fusion

```
for i=1 to N do
    A[i] = B[i] + 1
    C[i] = A[i] / 2
    D[i] = 1 / C[i+1]
endfor
```

Is this correct?
Cannot fuse
the third loop

Code Optimization

- Optimization framework
  - Control flow graph
    - To facilitate data flow analysis for optimization
    - To facilitate loop identification
  - Data flow analysis
    - Reachability analysis, copy propagation, expression propagation
    - Constant folding
    - Liveliness analysis, dead code elimination
  - Loop optimization
  - Function optimization
  - Alias analysis (pointers)