Introduction to
Machine-Independent Optimizations - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- What is code optimization?
- Illustrations of code optimizations
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations
Intermediate code generation process introduces many inefficiencies
- Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.

Code optimization removes such inefficiencies and improves code

Improvement may be time, space, or power consumption

It changes the structure of programs, sometimes of beyond recognition
- Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.

Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
Examples of Machine-Independent Optimizations

- Global common sub-expression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Partial redundancy elimination
- Loop unrolling
- Function inlining
- Tail recursion removal
- Vectorization and Concurrentization
- Loop interchange, and loop blocking
Bubble Sort

```c
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted
GCSE Conceptual Example

Demonstrating the need for repeated application of GCSE
GCSE on Running Example - 1

Y.N. Srikant

Introduction to Optimizations
Y.N. Srikant  Introduction to Optimizations

**GCSE on Running Example - 2**

- **B1:** \(i = 0\)
- **B2:** \(t1 = i > 1\)  
  *if \(t1\) goto B9*  
  *false*
- **B3:** \(j = 0\)  
  \(t2 = i - 1\)  
  *stop*
- **B9:**
- **B4:** \(t3 = j < t2\)  
  *if \(t3\) goto B8*  
  *false*
- **B5:** \(t4 = 4^*j\)  
  \(t5 = a[t4]\)  
  \(t6 = j + 1\)  
  \(t7 = 4^*t6\)  
  \(t8 = a[t7]\)  
  \(t9 = t5 > t8\)  
  *if \(t9\) goto B7*  
  *true*
- **B7:** \(t20 = t6\)  
  \(j = t20\)  
  *goto B4*  
  *false*
- **B8:**
- **B6:**
- **B10:** \(t10 = t4\)  
  \(t11 = a[t10]\)  
  \(temp = t11\)  
  \(t12 = t4\)  
  \(t13 = a + t12\)  
  \(t14 = t6\)  
  \(t15 = 4^*t14\)  
  \(t16 = a[t15]\)  
  \(*t13 = t16\)  
  \(t17 = t6\)  
  \(t18 = 4^*t17\)  
  \(t19 = a + t18\)  
  \(*t19 = temp\)
Copy Propagation on Running Example

```
B1  i = 0

B2  t1 = i>1
    if !t1 goto B9
    true
    false

B3  j = 0
    t2 = i-1
    stop

B4  t3 = j<t2
    if !t3 goto B8
    true
    false

B5  t4 = 4*j
    t5 = a[t4]
    t6 = j+1
    t7 = 4 * t6
    t8 = a[t7]
    t9 = t5 > t8
    if !t9 goto B7
    true
    false

B6  t11 = a[t4]
    temp = t11
    t13 = a + t4
    t15 = 4 * t6
    t16 = a[t15]
    *t13 = t16
    t18 = 4 * t6
    t19 = a + t18
    *t19 = temp

B7  j = t6
    goto B4
```
Constant Propagation and Folding Example

Before constant propagation:

Start

a = 10
b = 20
if b == 20 goto B3

yes

a = 30

no

d = a+5

Stop

After constant propagation and folding:

Start

a = 10
b = 20

a = 30

d = a+5

Stop

a = 30

d = 35

Stop
Loop Invariant Code Motion Example

Before LIV code motion

L1:  t2 = i > 100
     if t2 goto L2
     t1 = t1 - 2
     t3 = addr(a)
     t4 = t3 - 4
     t5 = 4 * i
     t6 = t4 + t5
     *t6 = t1
     i = i + 1
     goto L1

L2: 

After LIV code motion

L1:  t2 = i > 100
     if t2 goto L2
     t1 = t1 - 2
     t3 = addr(a)
     t4 = t3 - 4
     t5 = 4 * i
     t6 = t4 + t5
     *t6 = t1
     i = i + 1
     goto L1

L2: 

Y.N. Srikant  Introduction to Optimizations
Strength Reduction

Before strength reduction for t5

L1: t2 = i > 100
    if t2 goto L2
    t1 = t1 - 2
    t5 = 4 * i
    t6 = t4 + t5
    *t6 = t1
    i = i + 1
    goto L1

L2:

After strength reduction for t5 and copy propagation

L1: t2 = i > 100
    if t2 goto L2
    t1 = t1 - 2
    t4 = t3 - 4
    t7 = 4
    t3 = addr(a)
    t6 = t4 + t7
    *t6 = t1
    i = i + 1
    t7 = t7 + 4
    goto L1

L2:
Induction Variable Elimination

Before induction variable elimination (i)

L1: \[ t2 = i > 100 \]
\[ \text{if } t2 \text{ goto } L2 \]
\[ t1 = t1 - 2 \]
\[ t6 = t4 + t7 \]
\[ *t6 = t1 \]
\[ i = i + 1 \]
\[ t7 = t7 + 4 \]
\[ \text{goto } L1 \]

L2:

After eliminating i and replacing it with t7

L1: \[ t2 = t7 > 400 \]
\[ \text{if } t2 \text{ goto } L2 \]
\[ t1 = t1 - 2 \]
\[ t6 = t4 + t7 \]
\[ *t6 = t1 \]
\[ t7 = t7 + 4 \]
\[ \text{goto } L1 \]
Partial Redundancy Elimination

(a)

1. \( a = c \)
   \( x = a + b \)

2. \( y = a + b \)
   \( b = d \)

(b)

1. \( a = c \)
   \( h = a + b \)
   \( x = h \)

2. \( h = a + b \)

3. \( y = h \)
   \( b = d \)
for (i = 0; i<N; i++) { S_1(i); S_2(i); }

for (i = 0; i+3 < N; i+=3) {
    S_1(i); S_2(i);
    S_1(i+1); S_2(i+1);
    S_1(i+2); S_2(i+2);
}

// remaining few iterations, 1,2, or 3:
// (((N-1) mod 3)+1)
for (k=i; k<N; k++) { S_1(k); S_2(k); }
Unrolling While and Repeat loops

while (C) { S₁; S₂; }

repeat { S₁; S₂; } until C;

while (C) {
    S₁; S₂;
    if (!C) break;
    S₁; S₂;
    if (!C) break;
    S₁; S₂;
}

repeat {
    S₁; S₂;
    if (C) break;
    S₁; S₂;
    if (C) break;
    S₁; S₂;
} until C;
int find_greater(int A[10], int n) {
    int i;
    for (i=0; i<10; i++) {
        if (A[i] > n) return i;
    }
}
// inlined call: x = find_greater(Y, 250);
int new_i, new_A[10];
new_A = Y;
for (new_i=0; new_i<10; new_i++) {
    if (new_A[new_i] > 250) {
        x = new_i; goto exit;
    }
}
exit:
void sum (int A[], int n, int* x) {
    if (n==0) *x = *x+ A[0]; else {
        *x = *x+A[n]; sum(A, n-1, x);
    }
}

// after removal of tail recursion
void sum (int A[], int n, int* x) {
    while (true) { if (n==0) {*x=*x+ A[0]; break;}
        else{ *x=*x + A[n]; n=n-1; continue;}
    }
}
for $I = 1$ to 100 do {
    $X(I) = X(I) + Y(I)$
}

can be converted to

$X(1:100) = X(1:100) + Y(1:100)$

or

forall $I = 1$ to 100 do $X(I) = X(I) + Y(I)$
for \( I = 1 \) to 100 do {
    \( X(I+1) = X(I) + Y(I) \)
}

cannot be converted to

\( X(2:101) = X(1:100) + Y(1:100) \)

or equivalent concurrent code

because of dependence as shown below

\( X(2) = X(1) + Y(1) \)
\( X(3) = X(2) + Y(2) \)
\( X(4) = X(3) + Y(3) \)

...
Loop Interchange for parallelizability

```
for l = 1 to N do {
    for j = 1 to N do {
        S:  A(l+1,j) = A(l,j) * B(l,j) + C(l,j)
    }
}
```

Outer loop is not parallelizable, but inner loop is
Less work per thread

```
for j = 1 to N do {
    for l = 1 to N do {
        S:  A(l+1,j) = A(l,j) * B(l,j) + C(l,j)
    }
}
```

Outer loop is parallelizable but inner loop is not
More work per thread

```
forall j = 1 to N do {
    for l = 1 to N do {
        S:  A(l+1,j) = A(l,j) * B(l,j) + C(l,j)
    }
}
```
{ for (i = 0; i < N; i++)
    for (j=0; j < M; j++)
        A[j,l] = B[i] + C[j];
}

// Loop after blocking
{ for (ii = 0; ii < N; ii = ii+64)
    for (jj = 0; jj < M; jj = jj+64)
        for (i = ii; i < ii+64; i++)
            for (j=jj; j < jj+64; j++)
                A[j,l] = B[i] + C[j];
}
Fundamentals of Data-flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Data-flow analysis

- These are techniques that derive information about the flow of data along program execution paths.
- An execution path (or path) from point $p_1$ to point $p_n$ is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n-1$, either:
  1. $p_i$ is the point immediately preceding a statement and $p_{i+1}$ is the point immediately following that same statement, or
  2. $p_i$ is the end of some block and $p_{i+1}$ is the beginning of a successor block.
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path.
- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts.
- No analysis is necessarily a perfect representation of the state.
Uses of Data-flow Analysis

- Program debugging
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*

- Program optimizations
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.
A data-flow value for a program point represents an abstraction of the set of all possible program states that can be observed for that point.

The set of all possible data-flow values is the domain for the application under consideration.

- Example: for the reaching definitions problem, the domain of data-flow values is the set of all subsets of definitions in the program.
- A particular data-flow value is a set of definitions.

IN[s] and OUT[s]: data-flow values before and after each statement s.

The data-flow problem is to find a solution to a set of constraints on IN[s] and OUT[s], for all statements s.