Introduction to Machine-Independent Optimizations Part 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Outline of the Lecture

- What is code optimization?
- Types of code optimizations
- Illustrations of code optimizations

We discussed topics 1,2, and parts of topic 3 in part 1 of the lecture.

Machine-independent Code Optimization

- 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)

Unrolling a For-loop

```
for (i = 0; i<N; i++) { S1(i); S2(i); }

for (i = 0; i+3 < N; i+=3) {
    S1(i); S2(i);
    S1(i+1); S2(i+1);
    S1(i+2); S2(i+2);
}
// remaining few iterations, needed if N-1 is
// not a multiple of 3
for (k=i; k<N; i++) { S1(k); S2(k); }
```

Unrolling While and Repeat loops

```
while (C) { S1; S2; }

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

```
repeat { S1; S2; } until C;

repeat {
   S1; S2;
   if (C) break;
   S1; S2;
   if (C) break;
   S1; S2;
} until C;
```

Function Inlining

```
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:
```

Tail Recursion Removal

```
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;}
  }
}
```

Vectorization and Concurrentization Example 1

```
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)
```

Vectorization Example 2

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

Outer loop is not parallelizable, but inner loop is

Less work per thread

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

Outer loop is parallelizable but inner loop is not

More work per thread

```
forall J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Loop Blocking

```
{ for (i = 0; i < N; i++)
    for (j=0; j < M; j++)
      A[j,I] = 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,I] = B[i] + C[j];
}
```