вынос инвариантов/Loop-invariant code motion /

Loop-invariant code motion The loop-invariant code motion optimization recognizes instructions inside a loop whose results do not change and then moves the instructions outside the loop. This optimization ensures that the invariant code is only executed once.
For example, the C/C++ code:

```
x = z;
for(i=0; i<10; i++)
        a[i] = 4 * x + i;
```

Листинг 1

```
x = z;
t1 = 4 * x;
for(i=0; i<10; i++)
        a[i] = t1 + i;
```

Листинг 2

gcc{
-ftree-lim
Perform loop invariant motion on trees. This pass moves only invartiants that would be hard to handle on rtl level (function calls, operations that expand to nontrivial sequences of insns). With -funswitch-loops it also moves operands of conditions that are invariant out of the loop, so that we can use just trivial invariantness analysis in loop unswitching. The pass also includes store motion.

-fmove-loop-invariants
Enables the loop invariant motion pass in the new loop optimizer. Enabled at level -O1

lim-expensive
The minimum cost of an expensive expression in the loop invariant motion.
}

Strip mining

Strip mining is a fundamental +O3 transformation. In and of itself, strip mining is not profitable. However, it is used by loop blocking, loop unroll and jam, and, in a sense, by parallelization.
Strip mining involves splitting a single loop into a nested loop. The resulting inner loop iterates over a section or strip of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's strip length. Consider the following Fortran code:

```
DO I = 1, 10000
        A(I) = A(I) * B(I)
ENDDO
```

Strip mining this loop using a strip length of 1000 yields the following loop nest:

```
DO IOUTER = 1, 10000, 1000
        DO ISTRIP = IOUTER, IOUTER+999
                A(ISTRIP) = A(ISTRIP) * B(ISTRIP)
        ENDDO
ENDDO
```

In this loop, the strip length integrally divides the number of iterations, so the loop is evenly split up. If the iteration count was not an integral multiple of the strip length, for example, if I went from 1 to 10500 rather than 1 to 10000, the final iteration of the strip loop would execute 500 iterations instead of 1000.

Ветвления внутри циклов всегда нежелательны, а на старших моделях микропроцессоров Intel – особенно (кстати, большинство Си компиляторов платформы CONVEX вообще отказываются компилировать программы с ветвлениями внутри циклов).

Уникальной особенностью компилятора Microsoft Visual C++ является его умение избавляться от некоторых типов внутрицикловых ветвлений. Алгоритм подобной оптимизации слишком сложен, чтобы быть описанным в рамках журнальной статьи, поэтому, просто рассмотрим пример кода до и после оптимизации, а любопытных отошлем к книгам "Техника дизассемблирования программ" и "Техника оптимизации программ" Криса Касперски.

Peephole optimizations

// msvc: no
// icl:no
// gcc:no

A peephole optimization is a machine-dependent optimization that makes a pass through low-level assembly-like instruction sequences of the program, applying patterns to a small window (peephole) of code looking for optimization opportunities. The optimizations performed are:
Changing the addressing mode of instructions so they use shorter sequences
Replacing low-level assembly-like instruction sequences with faster (usually shorter) sequences, and removing redundant register loads and stores

Cloning within a single source file

Cloning is the replacement of a call to a routine by a call to a clone of that routine. The clone is optimized differently than the original routine. Cloning can expose additional opportunities for interprocedural optimization. At +O3, cloning is performed within a file; at +O4, it is performed across files. Cloning is enabled by default; it can be disabled by specifying the +Onoinline command-line option.

устранение зависимостей loop-carried dependences (LCD)

// msvc: yes
// icl: yes
// gcc: yes

```
for(i=1; i<n; i++)
        for (j=1; j<m; j++)
                a[j][i]=a[j-1][i-1] + a[j+1][i-1];
```

Листинг 3

```
for(j=1; j<m; j++)
        for (i=1;i <n; i++)
                a[j][i]=a[j+1][i-1] + a[j-1][i-1];
```

Листинг 4

Loop tiling

// msvc: no
// icl: no
// gcc: no
Loop tiling is a powerful high-level loop optimization technique useful for memory hierarchy optimization [14]. See the matrix multiplication program fragment:

```
for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
                for (k = 0; k < N; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Два последовательные обращения к одному и тому же элементу массива "a", происходят через каждые N операций умножения-и-сложения. Два последовательные обращения к одному и тому же элементу массива "b", происходят через каждые N*N операций умножения-и-сложения. Два последовательные обращения к одному и тому же элементу массива "c", происходят через каждую операцию умножения-и-сложения.

Two successive references to the same element of a are separated by N multiply-and-sum operations. Two successive references to the same element of b are separated by N2 multiply-and-sum operations. Two successive references to the same element of c are separated by 1 multiply-and-sum operation. For the case when N is large, references to a and b exhibits little locality and the frequent data fetching from memory results in high power consumption. Tiling the loop will transforme it to:

```
for (i = 0; i < N; i+=T)
        for (j = 0; j < N; j+=T)
                for (k = 0; k < N; k+=T)
                        for (ii = i; ii < min(i+T, N); ii++)
                                for (jj = j; jj < min(j+T, N); jj++)
                                        for (kk = k; kk < min(k+T, N); kk++)
                                                c[ii][jj] = c[ii][jj] + a[ii][kk] * b[kk][jj];
```

Blocking example: simple loop

// msvc:no
// icl: no
// gcc: no

In order to exploit reuse in more realistic examples that manipulate arrays that will not all fit in the cache, the compiler can apply the blocking transformation.
Consider the following Fortran example:

```
REAL*8 A(1000,1000),B(1000,1000)
REAL*8 C(1000),D(1000)
COMMON /BLK2/ A, B, C
.
DO J = 1, 1000
        DO I = 1, 1000
                A(I,J) = B(J,I) + C(I) + D(J)
        ENDDO
ENDDO
```

Now the compiler moves the outer strip loop (IOUT) outward as far as possible.

```
DO IOUT = 1, 1000, IBLOCK
        DO J = 1, 1000
                DO I = IOUT, IOUT+IBLOCK-1
                        A(I,J) = B(J,I) + C(I) + D(J)
                ENDDO
```

```
        ENDDO
ENDDO
```

```
do i = 1,n
do j = 1,m
B(j,i) = A(i,j)
end do
end do
```

```
do j1 = 1,n-nb+1,nb
j2 = min(j1+nb-1,n)
do i1 = 1,m-nb+1,nb
i2 = min(i1+nb-1,m)
do i = i1, i2
do j = j1, j2
B(j,i) = A(i,j)
end do
end do
end do
end do
```

_

```
DO I=1,N
…. (I) ….
ENDDO
```

```
DO i1=1,N,nb
  DO I=i1,min(i1+nb-1,N)
:. (I) :.
ENDDO ENDDO
```

Loop Skewing

```
// msvc: no
// icl: no
// gcc: no
Original Code
int i, s[N], a[N][N];
for(i=1; i<N-1; i++;)
        for(j=1; j<N-1; j++;)
        {
                a[i,j] = (a[i-1,j] + a[i,j-1] + a[i+1,j] + a[i,j+1]) / 4;
        }

After Loop Skewing
int i, s[N], a[N][N];
for(i=1; i<N-1; i++;)
for(j=i+1; j<i+N-1; j++;)
{
        a[i,j-i] = (a[i-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i])/4;
}
" Skewed Space
```

" Both loop can now be parallelized
" A Loop Interchange is needed

```
DO I=1,3
DO J=1,3
A(I, 2*J) = J
END DO
END DO
```

Figure 7: Original loop

```
DO U=1,3
DO V=U + 1,U + 3
A(U, 2*(V-U)) = 2*(V-U)
END DO
END DO
```

Cache thrashing

сокращение сложности вычислений/Strength reduction of induction variables and constants

```
// msvc: yes
// icl: yes
// gcc: yes

for (a=0; a<0x69; a++) r[a] = a * n;
```

Листинг 5

```
>->

t1 = 0; for (i=0; i<25; i++) { r[i] = t1; t1 += k; }
```

замена умножения делением

```
for (i=0;i<n;i++)
b[i] = a[i]/s

rs=1/s
for (i=0;i<n;i=i+1)
{b[i] = a[i]*rs
b[i+1] = a[i+1]*rs}

// msvc:no
// icl:no
// gcc:no
```

снижение сложности вычислений Code Motion

```
// msvc: yes
// icl: yes, may be
// gcc: yes, may be

for (i = 0; i < n; i++)
```

```
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];


for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

"сокращение сложности"/Reduction in Strength

This optimization also replaces multiplication instructions with addition instructions wherever possible. For example, in the following C/C++ code:

```
for (i=0; i<0x69; i++)
{
       r[i] = i * k;
}

>>>

t1 = 0;
for (i=0; i<0x69; i++)
{
       r[i] = t1;
       t1 += k;
}
```

gcc{
-fstrength-reduce
  Perform the optimizations of loop strength reduction and elimination of iteration variables.
Enabled at levels -O2, -O3, -Os.
}