Microsoft Visual C++ Floating-Point Optimization

Visual Studio .NET 2003
26 out of 39 rated this helpful - Rate this topic

Eric Fleegal
Microsoft Corporation

Revised June 2004

Applies To
  Microsoft® Visual C++®

Summary: Get a handle on optimizing floating-point code using the Microsoft Visual C++ 2005 (formerly known as Visual C++ "Whidbey") method of managing floating-point semantics. Create fast programs while ensuring that only safe optimizations are performed on floating-point code. (33 printed pages)

Contents

Optimization of Floating-Point Code in C++

An optimizing C++ compiler not only translates source code into machine code, it arranges the machine instructions in such a way as to improve efficiency and/or reduce size. Unfortunately, many common optimizations are not necessarily safe when applied to floating-point computations. A good example of this can be seen with the following summation algorithm [1]:

```
float KahanSum( const float A[], int n )
{
   float sum=0, C=0, Y, T;
   for (int i=0; i<n; i++)
   {
      Y = A[i] - C;
      T = sum + Y;
      C = T - sum - Y;
      sum = T;
   }
   return sum;
}
```

This function adds `n` float values in the array vector `A`. Within the loop body, the algorithm computes a "correction" value which is then applied to the next step of the summation. This method greatly reduces cumulative rounding errors as compared to a simple summation, while retaining O(n) time complexity.

A naïve C++ compiler might assume that floating-point arithmetic follows the same algebraic rules as Real number arithmetic. Such a compiler might then erroneously conclude that

```
C = T - sum - Y ==> (sum+Y)-sum-Y ==> 0;
```

That is, that the perceived value of C is always a constant zero. If this constant value is then propagated into subsequent expressions, the loop body is reduced to a simple summation. To be precise,

```
Y = A[i] - C ==> Y = A[i]
T = sum + Y ==> T = sum + A[i]
sum = T ==> sum = sum + A[i]
```

Thus, to the naïve compiler, a logical transformation of the **KahanSum** function would be:

```
float KahanSum( const float A[], int n )
{
   float sum=0; // C, Y & T are now unused
   for (int i=0; i<n; i++)
      sum = sum + A[i];
   return sum;
}
```

Although the transformed algorithm is faster, *it is not at all an accurate representation of the programmer's intention*. The carefully crafted error correction has been removed entirely, and we're left with a simple, direct summation algorithm with all its associated error.

Of course a sophisticated C++ compiler would know that algebraic rules of Real arithmetic do not generally apply to floating-point arithmetic. However, even a sophisticated C++ compiler might still incorrectly interpret the programmer's intention.

Consider a common optimization that attempts to hold as many values in registers as possible (called "enregistering" a value). In the KahanSum example, this optimization might attempt to enregister the variables `C`, `Y` and `T` since they're used only within the loop body. If the register precision is 52bits (double) instead of 23bits (single), this optimization effectively type promotes `C`, `Y` and `T` to type `double`. If the `sum` variable is not likewise enregistered, it will remain encoded in single precision. This transforms the semantics of **KahanSum** to the following

```
float KahanSum( const float A[], int n )
{
   float sum=0;
   double C=0, Y, T; // now held in-register
   for (int i=0; i<n; i++)
   {
      Y = A[i] - C;
      T = sum + Y;
      C = T - sum - Y;
      sum = (float) T;
   }
   return sum;
}
```

Although Y, T and C are now computed at a higher precision, this new encoding may produce a less accurate result depending on the values in **A[]**. Thus even seemingly harmless optimizations may have negative consequences.

These kinds of optimization problems aren't restricted to "tricky" floating-point code. Even simple floating-point algorithms can fail when optimized incorrectly. Consider a simple, direct-summation algorithm:

```
float Sum( const float A[], int n )
{
   float sum=0;
   for (int i=0; i<n; i++)
      sum = sum + A[i];
   return sum;
}
```

Because some floating-point units are capable of performing multiple operations simultaneously, a compiler might choose to engage a *scalar reduction* optimization. This optimization effectively transforms the simple **Sum** function from above into the following:

```
float Sum( const float A[], int n )
{
   int n4 = n-n%4; // or n4=n4&(~3)
   int i;
   float sum=0, sum1=0, sum2=0, sum3=0;
   for (i=0; i<n4; i+=4)
   {
      sum = sum + A[i];
      sum1 = sum1 + A[i+1];
      sum2 = sum2 + A[i+2];
      sum3 = sum3 + A[i+3];
   }
   sum = sum + sum1 + sum2 + sum3;
   for (; i<n; i++)
      sum = sum + A[i];
   return sum;
}
```

The function now maintains four separate summations, which can be processed simultaneously at each step. Although the optimized function is now much faster, the optimized results can be quite different from the non-optimized results. In making this change, the compiler assumed associative floating-point addition; that is, that these two expressions are equivalent (a+b)+c == a+(b+c). However, associativity does not always hold true for floating-point numbers. Instead of computing the summation as:

```
sum = A[0]+A[1]+A[2]+...+A[n-1]
```

the transformed function now computes the result as

```
sum = (A[0]+A[4]+A[8]+...)
      +(A[1]+A[5]+A[9]+...)
      +(A[2]+A[6]+A[10]+...)
      +(A[3]+A[7]+A[11]+...)
      +...
```

For some values of A[], this different ordering of addition operations may produce unexpected results. To further complicate matters, some programmers may choose to anticipate such optimizations and compensate for them appropriately. In this case, a program can construct the array `A` in a different order so that the optimized `sum` produces the expected results. Moreover, in many circumstances the accuracy of the optimized result may be "close enough". This is especially true when the optimization provides compelling speed benefits. Video games, for example, require as much speed as possible but don't often require highly accurate floating-point computations. Compiler makers must therefore provide a mechanism for programmers to control the often disparate goals of speed and accuracy.

Some compilers resolve the tradeoff between speed and accuracy by providing a separate "switch" for each type of optimization. This allows developers to disable optimizations that are causing changes to floating-point accuracy for their particular application. While this solution may offer a high degree of control over the compiler, it introduces several additional problems:

- It is often unclear which switches to enable or disable.
- Disabling any single optimization may adversely affect the performance of non floating-point code.
- Each additional switch incurs many new switch combinations; the number of combinations quickly becomes unwieldy.

So while providing separate switches for each optimization may seem appealing, using such compilers can be cumbersome and unreliable.

Many C++ compilers offer a "consistency" floating-point model, (through a /Op or /fltconsistency switch) which enables a developer to create programs compliant with strict floating-point semantics. When engaged, this model prevents the compiler from using most optimizations on floating-point computations while allowing those optimizations for non-floating-point code. The consistency model, however, has a dark-side. In order to return predictable results on different FPU architectures, nearly all implementations of /Op round intermediate expressions to the user specified precision; for example, consider the following expression:

```
float a, b, c, d, e;
. . .
a = b*c + d*e;
```

In order to produce consistent and repeatable results under /Op, this expression gets evaluated as if it were implemented as follows:

```
float x = b*c;
float y = d*e;
a = x+y;
```

The final result now suffers from single-precision rounding errors *at each step in evaluating the expression*. Although this interpretation doesn't strictly break any C++ semantics rules, it's almost never the best way to evaluate floating-point expressions. It is generally more desirable to *compute the intermediate results in as high as precision as is practical*. For instance, it would be better to compute the expression a=b*c+d*e in a higher precision as in,

```
double x = b*c;
double y = d*e;
double z = x+y;
a = (float)z;
```

or better yet

```
long double x = b*c;
long double y = d*e
long double z = x+y;
a = (float)z;
```

When computing the intermediate results in a higher precision, the final result is significantly more accurate. Ironically, by adopting a consistency model, the likelihood of error is increased precisely when the user is trying to reduce error by disabling unsafe optimizations. Thus the consistency model can seriously reduce efficiency while simultaneously providing no guarantee of increased accuracy. To serious numerical programmers, this doesn't seem like a very good tradeoff and is the primary reason that the model is not generally well received.

Beginning with version 8.0 (Visual C++® 2005), the Microsoft C++ compiler provides a much better alternative. It allows programmers to select one of three general floating-point modes: fp:precise, fp:fast and fp:strict.

- Under **fp:precise**, only safe optimizations are performed on floating-point code and, unlike /Op, intermediate computations are consistently performed at the highest *practical* precision.
- **fp:fast** mode relaxes floating-point rules allowing for more aggressive optimization at the expense of accuracy.
- **fp:strict** mode provides all the general correctness of fp:precise while enabling fp-exception semantics and preventing illegal transformations in the presence of FPU environment changes (e.g. changes to the register precision, rounding direction etc).

Floating-point exception semantics can be controlled independently by either a command-line switch or a compiler pragma; by default, floating-point exception semantics are disabled under fp:precise and enabled under fp:strict. The compiler also provides control over FPU environment sensitivity and certain floating-point specific optimizations, such as contractions. This straight-forward model gives developers a great deal of control over the compilation of floating-point code without the burden of too many compiler switches or the prospect of undesirable side-effects.

The fp:precise Mode for Floating-Point Semantics

The default floating-point semantics mode is fp:precise. When this mode is selected, the compiler strictly adheres to a set of safety rules when optimizing floating-point operations. These rules permit the compiler to generate efficient machine code while maintaining the accuracy of floating-point computations. To facilitate the production of fast programs, the fp:precise model disables floating-point exception semantics (although they can be explicitly enabled). Microsoft has selected fp:precise as the default floating-point mode because it creates both fast and accurate programs.

To explicitly request the fp:precise mode using the command-line compiler, use the -fp:precise switch:

```
cl -fp:precise source.cpp
```

This instructs the compiler to use fp:precise semantics when generating code for the source.cpp file. The fp:precise model can also be invoked on a function-by-function basis using the float_control compiler pragma.

Under the fp:precise mode, the compiler never performs any optimizations that perturb the accuracy of floating-point computations. The compiler will always round correctly at assignments, typecasts and function calls, and intermediate rounding will be consistently performed at the same precision as the FPU registers. Safe optimizations, such as contractions, are enabled by default. Exception semantics and FPU environment sensitivity are disabled by default.

fp:precise Floating-Point Semantics	Explanation
Rounding Semantics	Explicit rounding at assignments, typecasts, and function calls Intermediate expressions will be evaluated at register precision
Algebraic Transformations	Strict adherence to non-associative, non-distributive floating-point algebra, unless a transformation is guaranteed to always produce the same results.
Contractions	Permitted by default (see also The fp_contract Pragma)
Order of Floating-point Evaluation	The compiler may reorder the evaluation of floating-point expressions provided that the final results are not altered.
FPU Environment Access	Disabled by default (see also The fpenv_access Pragma). The default precision and rounding mode is assumed.
Floating-point Exception Semantics	Disabled by default (see also fp:except switch)

Rounding Semantics for Floating-Point Expressions Under fp:precise

The fp:precise model always performs intermediate computations at the highest *practical* precision, explicitly rounding only at certain points in expression evaluation. Rounding to the user-specified precision *always* occurs in four places: (a) when an assignment is made, (b) when a typecast is performed, (c) when a floating-point value is passed as an argument to a function and (d) when a floating-point value is returned from a function. Because intermediate computations are always performed at register precision, the accuracy of intermediate results is platform dependant (though precision will always be at least as accurate as the user specified precision).

Consider the assignment expression in the following code. The expression on the right-hand side of the assignment operator '=' will be computed at register precision, and then explicitly rounded to the type of the left-hand side of the assignment.

```
float a, b, c, d;
double x;
...
x = a*b + c*d;
```

is computed as

```
float a, b, c, d;
double x;
...
register tmp1 = a*b;
register tmp2 = c*d;
register tmp3 = tmp1+tmp2;
x = (double) tmp3;
```

To explicitly round an intermediate result, introduce a typecast. For example, if the previous code is modified by adding an explicit typecast, the intermediate expression (c*d) will be rounded to the type of the typecast.

```
float a, b, c, d;
double x;
. . .
x = a*b + (float)(c*d);
```

is computed as

```
float a, b, c, d;
double x;
. . .
register tmp1 = a*b;
float tmp2 = c*d;
register tmp3 = tmp1+tmp2;
x = (double) tmp3
```

One implication of this rounding method is that some seemingly equivalent transformations don't actually have identical semantics. For instance, the following transformation splits a single assignment expression into two assignment expressions.

```
float a, b, c, d;
. . .
a = b*(c+d);
```

is NOT equivalent to

```
float a, b, c, d;
. . .
a = c+d;
a = b*a;
```

Likewise:

```
a = b*(c+d);
```

is NOT equivalent to

```
a = b*(a=c+d);
```

These encodings do not have equivalent semantics because the second encodings have each introduced an additional assignment operation, and hence an additional rounding point.

When a function returns a floating-point value, the value will be rounded to the type of the function. When a floating-point value is passed as a parameter to a function, the value will be rounded to the type of the parameter. For example:

```
float sumsqr(float a, float b)
{
   return a*a + b*b;
}
```

is computed as

```
float sumsqr(float a, float b)
{
    register tmp3 = a*a;
    register tmp4 = b*b;
    register tmp5 = tmp3+tmp4;
    return (float) tmp5;
}
```

Likewise:

```
float w, x, y, z;
double c;
...
c = symsqr(w*x+y, z);
```

is computed as

```
float x, y, z;
```

```
double c;
...
register tmp1 = w*x;
register tmp2 = tmp1+y;
float tmp3 = tmp2;
c = symsqr( tmp3, z);
```

Architecture-specific rounding under fp:precise

Processor	Rounding Precision for Intermediate Expressions
x86	Intermediate expressions are computed at the default 53-bit precision with an extended range provided by a 16-bit exponent. When these 53:16 values are "spilled" to memory (as can happen during a function call), the extended exponent range will be narrowed to 11-bits. That is, spilled values are cast to the standard double precision format with only an 11-bit exponent. A user may switch to extended 64-bit precision for intermediate rounding by altering the floating-point control word using **_controlfp** and by enabling FPU environment access (see The fpenv_access Pragma). However, when extended precision register-values are spilled to memory, the intermediate results will still be rounded to double precision. This particular semantic is subject to change.
ia64	Intermediate expressions are always computed at 64-bit precision (that is, extended precision).
amd64	FP semantics on amd64 are somewhat different from other platforms. For performance reasons, intermediate operations are computed at the widest precision of either operand instead of at the widest precision available. To force computations to be computed using a wider precision than the operands, users need to introduce a cast operation on at least one operand in a sub-expression. This particular semantic is subject to change.

Algebraic Transformations Under fp:precise

When the fp:precise mode is enabled, the compiler will never perform algebraic transformations *unless the final result is provably identical*. Many of the familiar algebraic rules for Real number arithmetic do not always hold for floating-point arithmetic. For example, the following expressions are equivalent for Reals, but not necessarily for Floats.

Form	Description
(a+b)+c = a+(b+c)	Associative rule for addition
(a*b)*c = a*(b*c)	Associative rule for multiplication
a*(b+c) = a*b + b*c	Distribution of multiplication over addition
(a+b)(a-b) = a*a-b*b	Algebraic Factoring
a/b = a*(1/b)	Division by multiplicative inverse
a*1.0 = a	Multiplicative identity

As depicted in the introduction example with the function KahanSum, the compiler might be tempted to perform various algebraic transformations in order to produce considerably faster programs. Although optimizations dependent on such algebraic transformations are almost always incorrect, there are occasions for which they are perfectly safe. For instance, it is sometimes desirable to replace division by a *constant* value with multiplication by the multiplicative-inverse of the constant:

```
const double four = 4.0;
double a, b;
...

a = b/four;
```

May be transformed into

```
const double four = 4.0;
const double tmp0 = 1/4.0;
double a, b;
...
a = b*tmp0;
```

This is a safe transformation because the optimizer can determine at compile time that x/4.0 == x*(1/4.0) for all floating-point values of x, including infinities and NaN. By replacing a division operation with a multiplication, the compiler can save several cycles—especially on FPUs that don't directly implement division, but require the compiler to generate a combination of reciprocal-approximation and multiply-add instructions. The compiler may perform such an optimization under fp:precise only when the replacement multiplication produces the exact same result as the division. The compiler may also perform trivial transformations under fp:precise, provided the results are identical. These include:

Form	**Description**
(a+b) == (b+a)	Commutative rule for addition
(a*b) == (b*a)	Commutative rule for multiplication
1.0*x*y == x*1.0*y == x*y*1.0 == x*y	Multiplication by 1.0
x/1.0*y == x*y/1.0 == x*y	Division by 1.0
2.0*x == x+x	Multiplication by 2.0

Contractions Under fp:precise

A key architectural feature of many modern floating-point units is the ability to perform a multiplication followed by an addition as a single operation with no intermediate round-off error. For example, Intel's Itanium architecture provides instructions to combine each of these ternary operations, (a*b+c), (a*b-c) and (c-a*b), into a single floating-point instruction (fma, fms and fnma respectively). These single instructions are faster than executing separate multiply and add instructions, and are more accurate since there is no intermediate rounding of the product. This optimization can significantly speed up functions containing several interleaved multiply and add operations. For example, consider the following algorithm which computes the dot-product of two n-dimensional vectors.

```
float dotProduct( float x[], float y[], int n )
```

```
{
   float p=0.0;
   for (int i=0; i<n; i++)
      p += x[i]*y[i];
   return p;
}
```

This computation can be performed a series of multiply-add instructions of the form `p = p + x[i]*y[i]`.

The contraction optimization can be independently controlled using the **fp_contract** compiler pragma. By default, the fp:precise model allows for contractions since they improve both accuracy and speed. Under fp:precise, the compiler will never contract an expression with explicit rounding.
Examples

```
float a, b, c, d, e, t;
...
d = a*b + c;          // may be contracted
d += a*b;             // may be contracted
d = a*b + e*d;        // may be contracted into a mult followed by a mult-
add
etc...

d = (float)a*b + c;   // won't be contracted because of explicit rounding

t = a*b;              // (this assignment rounds a*b to float)
d = t + c;            // won't be contracted because of rounding of a*b
```

Order of Floating-Point Expression Evaluation Under fp:precise

Optimizations that preserve the order of floating-point expression evaluation are always safe and are therefore permitted under the fp:precise mode. Consider the following function which computes the dot product of two n-dimensional vectors in single precision. The first code block below is the original function as it might be encoded by a programmer, followed by the same function after a partial loop-unrolling optimization (changes *italicized*).

```
//original function
float dotProduct( float x[], float y[],
                  int n )
{
   float p=0;
   for (int i=0; i<n; i++)
      p += x[i]*y[i];
   return p;
}


//after a partial loop-unrolling
float dotProduct( float x[], float y[],
                  int n )
{
int n4= n/4*4; // or n4=n&(~3);
float p=0;
int i;
for (i=0; i<n4; i+=4)
{
p+=x[i]*y[i];
p+=x[i+1]*y[i+1];
p+=x[i+2]*y[i+2];
p+=x[i+3]*y[i+3];
}

// last n%4 elements
for (; i<n; i++)
p+=x[i]*y[i];

return p;
}
```

The primary benefit of this optimization is that it reduces the number of conditional loop-branching by as much as 75%. Also, by increasing the number of operations within the loop body, the compiler may now have more opportunities to optimize further. For instance, some FPUs may be able to perform the multiply-add in p+=x[i]*y[i] while simultaneously fetching the values for x[i+1] and y[i+1] for use in the next step. This type of optimization is perfectly safe for floating-point computations because it preserves the order of operations.

It's often advantageous for the compiler to reorder entire operations in order to produce faster code. Consider the following code:

```
double a, b, c, d;
double x, y, z;
...
x = a*a*a + b*b*b + c*c*c;
...
y = a*a + b*b + c*c;
...
z = a + b + c;
```

C++ semantic rules indicate that the program should produce results *as if* it first computed x, then y and finally z. Suppose the compiler has only four available floating-point registers. If the compiler is forced to compute x, y and z in order, it might choose to generate code with the following semantics:

```
double a, b, c, d;
double x, y, z;
register r0, r1, r2, r3;
...
// Compute x
r0 = a;         // r1 = a*a*a
r1 = r0*r0;
r1 = r1*r0;
r0 = b;         // r2 = b*b*b
r2 = r0*r0;
r2 = r2*r0;
r0 = c;         // r3 = c*c*c
r3 = r0*r0;
r3 = r3*r0;
r0 = r1 + r2;
r0 = r0 + r3;
```

```
x = r0;          // x = r1+r2+r3
. . .
// Compute y
r0 = a;          // r1 = a*a
r1 = r0*r0;
r0 = b;          // r2 = b*b
r2 = r0*r0;
r0 = c;          // r3 = c*c
r3 = r0*r0;
r0 = r1 + r2;
r0 = r0 + r3;
y = r0;          // y = r1+r2+r3
. . .
// Compute z
r1 = a;
r2 = b;
r3 = c;
r0 = r1 + r2;
r0 = r0 + r3;
z = r0;          // z = r1+r2+r3
```

There are several clearly redundant operations is this encoding (***italicized***). If the compiler strictly follows C++ semantic rules, this ordering is necessary because the program might access the FPU environment in-between each assignment. However, the default settings for fp:precise allow the compiler to optimize as though the program doesn't access the environment, allowing it to reorder these expressions. It's then free to remove the redundancies by computing the three values in reverse order, as follows:

```
double a, b, c, d;
double x, y, z;
register r0, r1, r2, r3;
...
// Compute z
r1 = a;
r2 = b;
r3 = c;
r0 = r1+r2;
r0 = r0+r3;
z = r0;
...
// Compute y
r1 = r1*r1;
r2 = r2*r2;
r3 = r3*r3;
r0 = r1+r2;
r0 = r0+r3;
y = r0;
...
// Compute x
r0 = a;
r1 = r1*r0;
r0 = b;
r2 = r2*r0;
r0 = c;
r3 = r3*r0;
r0 = r1+r2;
r0 = r0+r3;
x = r0;
```

This encoding is clearly superior, having reduced the number of fp-instructions by almost 40%. The results for x, y and z are the same as before, but computed with less overhead.

Under fp:precise, the compiler may also *interlace* common sub-expressions so as to produce faster code. For example, code to compute the roots of a quadratic equation might be written as follows:

```
double a, b, c, root0, root1;
...
root0 = (-b + sqrt(b*b-4*a*c))/(2*a);
root1 = (-b - sqrt(b*b-4*a*c))/(2*a);
```

Although these expressions only differ by a single operation, the programmer may have written it this way to guarantee that each root value will be computed in the highest practical precision. Under fp:precise, the compiler is free to interlace the computation of root0 and root1 to remove common sub-expressions without losing precision. For instance, the following has removed several redundant steps while producing the exact same answer.

```
double a, b, c, root0, root1;
...
register tmp0 = -b;
register tmp1 = sqrt(b*b-4*a*c);
register tmp2 = 2*a;
root0 = (tmp0+tmp1)/tmp2;
root1 = (tmp0-tmp1)/tmp2;
```

Other optimizations may attempt to move the evaluation of certain independent expressions. Consider the following algorithm which contains a conditional-branch within a loop-body.

```
vector<double> a(n);
double d, s;
. . .
for (int i=0; i<n; i++)
{
   if (abs(d)>1.0)
      s = s+a[i]/d;
   else
      s = s+a[i]*d;
}
```

The compiler may detect that the value of the expression (abs(d)>1) is invariant within the loop body. This enables the compiler to "hoist" the if statement outside the loop body, transforming the above code into the following:

```
vector<double> a(n);
double d, s;
. . .
if (abs(d)>1.0)
   for (int i=0; i<n; i++)
      s = s+a[i]/d;
else
   for (int i=0; i<n; i++)
      s = s+a[i]*d;
```

After the transformation, there is no longer a conditional branch in either of the loop bodies, thus greatly improving the overall performance of the loop. This type of optimization is perfectly safe because the evaluation of the expression `(abs(d)>1.0)` is independent of other expressions.

In the presence of FPU environment access or floating-point exceptions, these types of optimization are contraindicated because they change the semantic flow. Such optimizations are only available under the fp:precise mode because FPU environment access and floating-point exception semantics are disabled by default. Functions that access the FPU environment can explicitly disable such optimizations by using the **fenv_access** compiler pragma. Likewise, functions using floating-point exceptions should use the float_control(except…) compiler pragma (or use the /fp:except command line switch).

In summary, the fp:precise mode allows the compiler to reorder the evaluation of floating-point expressions on condition that the final results are not altered and that results are not dependant on the FPU environment or on floating-point exceptions.

FPU Environment Access Under fp:precise

When the fp:precise mode is enabled, the compiler assumes that a program does not access or alter the FPU environment. As stated earlier, this assumption enables the compiler to reorder or move floating-point operations to improve efficiency under fp:precise.

Some programs may alter the floating-point rounding-direction by using the **_controlfp** function. For instance, some programs compute upper and lower error bounds on arithmetic operations by performing the same computation twice, first while rounding towards negative infinity, then while rounding towards positive infinity. Since the FPU provides a convenient way to control rounding, a programmer may choose to change rounding mode by altering the FPU environment. The following code computes an exact error bound of a floating-point multiplication by altering the FPU environment.

```
double a, b, cLower, cUpper;
. . .
_controlfp( _RC_DOWN, _MCW_RC );    // round to -&infin;
cLower = a*b;
_controlfp( _RC_UP, _MCW_RC );    // round to +&infin;
cUpper = a*b;
_controlfp( _RC_NEAR, _MCW_RC );    // restore rounding mode
```

Under fp:precise, the compiler always assumes the default FPU environment, so the optimizer is free to ignore the calls to **_controlfp** and reduce the above assignments to `cUpper = cLower = a*b`; this would clearly yield incorrect results. To prevent such optimizations, enable FPU environment access by using the **fenv_access** compiler pragma.

Other programs may attempt to detect certain floating-point errors by checking the FPU's status word. For example, the following code checks for the divide-by-zero and inexact conditions

```
double a, b, c, r;
```

```
float x;
. . .
_clearfp();
r = (a*b + sqrt(b*b-4*a*c))/(2*a);
if (_statusfp() & _SW_ZERODIVIDE)
   handle divide by zero as a special case
_clearfp();
x = r;
if (_statusfp() & _SW_INEXACT)
   handle inexact error as a special case
etc...
```

Under fp:precise ,optimizations that reorder expression evaluation may change the points at which certain errors occur. Programs accessing the status word should enable FPU environment access by using the **fenv_access** compiler pragma.

See pragma fenv_access for more information

Floating-Point Exception Semantics Under fp:precise

By default, floating-point exception semantics are disabled under fp:precise. Most C++ programmers prefer to handle exceptional floating-point conditions without using system or C++ exceptions. Moreover, as stated earlier, disabling floating-point exception semantics allows the compiler greater flexibility when optimizing floating-point operations. Use either the fp:except switch or the fp_control pragma to enable floating-point exception semantics when using the fp:precise model.

See Also
Enabling floating-point Exception Semantics

The fp:fast Mode for Floating-Point Semantics

When the fp:fast mode is enabled, the compiler relaxes the rules that fp:precise uses when optimizing floating-point operations. This mode allows the compiler to further optimize floating-point code for speed at the expense of floating-point accuracy and correctness. Programs that do not rely on highly accurate floating-point computations may experience a significant speed improvement by enabling the fp:fast mode.

The fp:fast floating-point mode is enabled using a command-line compiler switch as follows:

```
cl -fp:fast source.cpp
   or
cl /fp:fast source.cpp
```

This example instructs the compiler to use fp:fast semantics when generating code for the source.cpp file. The fp:fast model can also be invoked on a function-by-function basis using the **float_control** compiler pragma.

See Also
The float_control Pragma

Under the fp:fast mode, the compiler may perform optimizations that alter the accuracy of floating-point computations. The compiler may not round correctly at assignments, typecasts or function calls, and intermediate rounding will not always be performed. Floating-point specific optimizations, such as contractions, are always enabled. Floating-point exception semantics and FPU environment sensitivity are disabled and unavailable.

fp:fast floating-point semantics	**Explanation**
Rounding Semantics	Explicit rounding at assignments, typecasts, and function calls may be ignored. Intermediate expressions may be rounded at less than register precision according to performance requirements.
Algebraic Transformations	The compiler may transform expressions according real-number associative, distributive algebra; these transformations are not guaranteed to be either accurate or correct.
Contractions	Always enabled; cannot be disabled by pragma **fp_contract**
Order of Floating-point Evaluation	The compiler may reorder the evaluation of floating-point expressions, even when such changes may alter the final results
FPU Environment Access	Disabled. Not available
Floating-point Exception Semantics	Disabled. Not available

Rounding Semantics for Floating-Point Expressions Under fp:fast

Unlike the fp:precise model, the fp:fast model performs intermediate calculations at the most convenient precision. Rounding at assignments, typecasts and function-calls may not always occur. For instance, the first function below introduces three single-precision variables (**C**, **Y** and **T**). The compiler may choose to enregister these variables, in effect of type promoting **C**, **Y** and **T** to double-precision.

Original function:

```
float KahanSum( const float A[], int n )
{
   float sum=0, C=0, Y, T;
   for (int i=0; i<n; i++)
   {
      Y = A[i] - C;
      T = sum + Y;
      C = T - sum - Y;
      sum = T;
   }
   return sum;
}
```

Variables enregistered:

```
float KahanSum( const float A[], int n )
{
   float sum=0;
   double C=0, Y, T; // now held in-register
   for (int i=0; i<n; i++)
   {
      Y = A[i] - C;
      T = sum + Y;
      C = T - sum - Y;
      sum = (float) T;
   }
   return sum;
}
```

In this example, fp:fast has subverted the intent of the original function. The final optimized result, held in the variable **sum**, may be quite perturbed from the correct result.

Under fp:fast, the compiler will typically attempt to maintain at least the precision specified by the source code. However, in some instances the compiler may choose to perform intermediate expressions at a *lower precision* than specified in the source code. For example, the first code block below calls a double precision version of the square-root function. Under fp:fast, the compiler may choose to replace the call to the double precision **sqrt** with a call to a single precision **sqrt** function. This has the effect of introducing additional lower-precision rounding at the point of the function call.

Original function

```
double sqrt(double)...
float a, b, c;
double d;
. . .
double d = a * b;
float c = sqrt((float) d);
```

Optimized function

```
float sqrtf(float)...
. . .
float a, b,c;
. . .
double d = a * b; // up-convert to double-precision by C/C++ promotion
rules
float tmp0 = (float) d; // down-convert to single-precision by explicit
cast
float c = sqrtf(tmp0); // square root computed in single-precision
```

Although less accurate, this optimization may be especially beneficial when targeting processors that provide single precision, intrinsic versions of functions such as **sqrt**. Just precisely when the compiler will use such optimizations is both platform and context dependant.

Furthermore, there is no guaranteed consistency for the precision of intermediate computations, which may be performed at any precision level available to the compiler. Although the compiler will attempt to maintain at least the level of precision as specified by the code, fp:fast allows the optimizer to downcast intermediate computations in order to produce faster or smaller machine code. For instance, the compiler may further optimize the code from above to round the intermediate multiplication to single precision.

```
float sqrtf(float)...
float a, b, c;
. . .
// in this simple example, the store to d is optimized away
float tmp1 = a * b; // converts have been removed
float c = sqrtf(tmp1); // square root computed in single-precision
```

This kind of additional rounding may result from using a lower precision floating-point unit, such as SSE2, to perform some of the intermediate computations. The accuracy of fp:fast rounding is therefore platform dependant; code that compiles well for one processor may not necessarily work well for another processor. It's left to the user to determine if the speed benefits outweigh any accuracy problems.

If fp:fast optimization is particularly problematic for a specific function, the floating-point mode can be locally switched to fp:precise using the **float_control** compiler pragma.

Algebraic Transformations Under fp:fast

The fp:fast mode enables the compiler to perform certain, unsafe algebraic transformations to floating point expressions. For example, the following unsafe optimizations may be employed under fp:fast.

Original Code	**Step #1**	**Step #2**
`double a, b, c;`	`double a, b, c;`	`double a, b, c;`
`double x, y, z;`	`double x, y, z;`	`double x, y, z;`
`. . .`	`. . .`	`. . .`
`y = (a+b);`	`y = (a+b);`	`y = (a+b);`
`z = y – a – b;`	`z = 0;`	`z = 0;`
`. . .`	`. . .`	`. . .`
`c = x – z;`	`c = x – 0;`	`c = x;`
`. . .`	`. . .`	`. . .`
`c = x*z;`	`c = x*0;`	`c = 0;`
`. . .`	`. . .`	`. . .`
`c = x-z;`	`c = x-0;`	`c = x;`
`. . .`	`. . .`	`. . .`

```
c = x+z;            c = x+0;        c = x;

. . .               . . .           . . .

c = z-x;            c = 0-x;        c = -x;
```

In step 1, the compiler observes that `z = y – a – b` is always equal to zero. Although this is technically an invalid observation, it is permitted under fp:fast. The compiler then propagates the constant value zero to every subsequent use of the variable **z**. In step 2, the compiler further optimizes by observing that x-0==x, x*0==0, etc. Again, even though these observations are not strictly valid, they are permitted under fp:fast. The optimized code is now much faster, but may also be considerably less accurate or even incorrect.

Any of the following (unsafe) algebraic rules may be employed by the optimizer when the fp:fast mode is enabled:

Form	Description
(a+b)+c = a+(b+c)	Associative rule for addition
(a*b)*c = a*(b*c)	Associative rule for multiplication
a*(b+c) = a*b + b*c	Distribution of multiplication over addition
(a+b)(a-b) = a*a-b*b	Algebraic Factoring
a/b = a*(1/b)	Division by multiplicative inverse
a*1.0 = a, a/1.0 = a	Multiplicative identity
a±0.0 = a, 0.0-a = -a	Additive identity
a/a = 1.0, a-a = 0.0	Cancellation

If fp:fast optimization is particularly problematic for a particular function, the floating-point mode can be locally switched to fp:precise using the **float_control** compiler pragma.

Order of Floating-Point Expression Evaluation Under fp:fast

Unlike fp:precise, fp:fast allows the compiler to reorder floating-point operations so as to produce faster code, even though the reordering may alter the final result. Thus, some optimizations under fp:fast may not preserve the intended order of expressions. For instance, consider the following function that computes the dot product of two n-dimensional vectors.

```
float dotProduct( float x[], float y[],
                  int n )
{
   float p=0;
   for (int i=0; i<n; i++)
      p += x[i]*y[i];
   return p;
}
```

Under fp:fast, the optimizer may perform a scalar reduction of the dotProduct function effectively transforming the function as follows:

```
float dotProduct( float x[], float y[],int n )
```

```
{
    int n4= n/4*4; // or n4=n&(~3);
    float p=0, p2=0, p3=0, p4=0;
    int i;

    for (i=0; i<n4; i+=4)
    {
        p+=x[i]*y[i];
        p2+=x[i+1]*y[i+1];
        p3+=x[i+2]*y[i+2];
        p4+=x[i+3]*y[i+3];
    }
    p+=p2+p3+p4;

    // last n%4 elements
    for (; i<n; i++)
    p+=x[i]*y[i];

    return p;
}
```

In the optimized version of the function four separate product-summations are taken simultaneously and then added together. This optimization can speed up the computation of the dotProduct by as much as a factor of four depending on the target processor, but the final result may be so inaccurate as to render it useless. If such optimizations are particularly problematic for single function or translation unit, the floating-point mode can be locally switched to fp:precise using the **float_control** compiler pragma.

The fp:strict Mode for Floating-Point Semantics

When the fp:strict mode is enabled, the compiler adheres to all the same rules that fp:precise uses when optimizing floating-point operations. This mode also enables floating-point exception semantics and sensitivity to the FPU environment and disables certain optimizations such as contractions. It is the strictest mode of operation.

The fp:strict floating-point mode is enabled using a command-line compiler switch as follows:

```
cl -fp:strict source.cpp
   or
cl /fp:strict source.cpp
```

This example instructs the compiler to use fp:strict semantics when generating code for the source.cpp file. The fp:strict model can also be invoked on a function-by-function basis using the **float_control** compiler pragma.

See also:
The float_control Pragma

Under the fp:strict mode, the compiler never performs any optimizations that perturb the accuracy of floating-point computations. The compiler will always round correctly at assignments, typecasts and function calls, and intermediate rounding will be consistently performed at the same precision as the FPU registers. Floating-point exception semantics and FPU environment sensitivity are enabled by default. Certain optimizations, such as contractions, are disabled because the compiler cannot guarantee correctness in every case.

fp:strict Floating-Point Semantics	**Explanation**
Rounding Semantics	Explicit rounding at assignments, typecasts, and function calls Intermediate expressions will be evaluated at register precision. Same as fp:precise
Algebraic Transformations	Strict adherence to non-associative, non-distributive floating-point algebra, unless a transformation is guaranteed to always produce the same results. Same as fp:precise
Contractions	Always disabled
Order of Floating-point Evaluation	The compiler will not reorder the evaluation of floating-point expressions
FPU Environment Access	Always enabled.
Floating-point Exception Semantics	Enabled by default.

Floating-Point Exception Semantics Under fp:strict

By default, floating-point exception semantics are enabled under the fp:strict model. To disable these semantics, use either the "`-fp:except-`" switch or introduce a `float_control(except, off)` pragma.

See also:
Enabling floating-point Exception Semantics
The float_control Pragma

The fenv_access Pragma

Usage:

```
#pragma fenv_access( [ on  | off ] )
```

The **fenv_access** pragma allows the compiler to make certain optimizations that might subvert FPU flag tests and FPU mode changes. When the state of **fenv_access** is disabled, the compiler can assume the default FPU modes are in effect and that FPU flags are not tested. By default, environment access is disable for the fp:precise mode, though it may be explicitly enabled using this pragma. Under fp:strict, **fenv_access** is always enabled and cannot be disabled. Under fp:fast, **fenv_access** is always disabled, and cannot be enabled.

As described in the fp:precise section, some programmers may alter the floating-point rounding-direction using the **_controlfp** function. For example, to compute the upper and lower error bounds on arithmetic operations, some programs perform the same computation twice, first while rounding towards negative infinity, then while rounding towards positive infinity. Since the FPU provides a convenient way to control rounding, a programmer may choose to change rounding mode by altering the FPU environment. The following code computes an exact error bound of a floating-point multiplication by altering the FPU environment.

```
double a, b, cLower, cUpper;
. . .
_controlfp( _RC_DOWN, _MCW_RC );    // round to -8
cLower = a*b;
_controlfp( _RC_UP, _MCW_RC );       // round to +8
cUpper = a*b;
_controlfp( _RC_NEAR, _MCW_RC );    // restore rounding mode
```

When disabled, the **fenv_access** pragma allows the compiler to assume the default FPU environment; thus the optimizer is free to ignore the calls to **_controlfp** and reduce the above assignments to `cUpper = cLower = a*b`. When enabled, however, **fenv_access** prevents such optimizations.

Programs may also check the FPU status word to detect certain floating-point errors. For example, the following code checks for the divide-by-zero and inexact conditions

```
double a, b, c, r;
float x;
. . .
_clearfp();
r = (a*b + sqrt(b*b-4*a*c))/(2*a);
if (_statusfp() & _SW_ZERODIVIDE)
   handle divide by zero as a special case
_clearfp();
x = (a*b + sqrt(b*b-4*a*c))/(2*a);
if (_statusfp() & _SW_INEXACT)
   handle inexact error as a special case
etc...
```

When **fenv_access** is disabled, the compiler might rearrange the execution order of the floating-point expressions, thus possibly subverting the FPU status checks. Enabling **fenv_access** prevents such optimizations.

The fp_contract Pragma

Usage:

```
#pragma fp_contract( [ on | off ] )
```

As described in the fp:precise section, contraction is a fundamental architectural feature for many modern floating-point units. Contractions provide the ability to perform a multiplication followed by an addition as a single operation with no intermediate round-off error. For example, Intel's Itanium architecture provides instructions to combine each of these ternary operations, (a*b+c), (a*b-c) and (c-a*b), into a single floating-point instruction (fma, fms and fnma respectively). These single instructions are faster than executing separate multiply and add instructions, and are more accurate since there is no intermediate rounding of the product. A contracted operation can computes the value of (a*b+c) *as if* both operations were computed to infinite precision, and then rounded to the nearest floating-point number. This optimization can significantly speed up functions containing several interleaved multiply and add operations. For example, consider the following algorithm which computes the dot-product of two n-dimensional vectors.

```
float dotProduct( float x[], float y[], int n )
{
   float p=0.0;
   for (int i=0; i<n; i++)
      p += x[i]*y[i];
   return p;
}
```

This computation can be performed a series of multiply-add instructions of the form `p = p + x[i]*y[i]`.

The **fp_contract** pragma specifies whether floating-point expressions can be contracted. By default, the fp:precise mode allows for contractions since they improve both accuracy and speed. Contractions are always enabled for the fp:fast mode. However, because contractions can subvert the explicit detection of error conditions, the **`fp_contract`** pragma is always disabled under the fp:strict mode. Examples of expressions that may be contracted when the **`fp_contract`** pragma is enabled:

```
float a, b, c, d, e, t;
...
d = a*b + c;         // may be contracted
d += a*b;            // may be contracted
d = a*b + e*d;       // may be contracted into a mult followed by a mult-
add
etc...

d = (float)a*b + c;  // won't be contracted because of explicit rounding

t = a*b;             // (this assignment rounds a*b to float)
d = t + c;           // won't be contracted because of rounding of a*b
```

The float_control Pragma

The /fp:precise, /fp:fast, /fp:strict and /fp:except switches control floating-point semantics on a file-by-file basis. The **float_control** pragma provides such control on a function-by-function basis.

Usage:

```
#pragma float_control(push)
#pragma float_control(pop)

#pragma float_control( precise, on | off [, push] )
#pragma float_control( except, on | off [, push] )
```

The pragmas float_control(push) and float_control(pop) respectively push and pop the current state of the floating-point mode and the exception option onto a stack. Note that the state of the fenv_access and **fp_contract** pragma are not affected by pragma float_control(push/pop).

Calling the pragma float_control(precise, on | off) will enable or disable precise-mode semantics. Likewise, the pragma float_control(except, on | off) will enable or disable exception semantics. Exception semantics can only be enabled when precise semantics are also enabled. When the optional push argument is present, the states of the float_control options are pushed prior to changing semantics.

Setting the Floating-Point Semantic Mode on a Function-by-Function Basis

The command-line switches are in fact shorthand for setting the four different floating-point pragmas. To explicitly choose a particular floating-point semantic mode on a function-by-function basis, select each of the four floating-point option pragmas as described in the following table:

	float_control(precise)	**float_control(except)**	**fp_contract**	**fenv_access**
-fp:strict	on	on	off	on
-fp:strict -fp:except-	on	off	off	on
-fp:precise	on	off	on	off
-fp:precise –fp:except	on	on	on	off
-fp:fast	off	off	on	off

For example, the following explicitly enables fp:fast semantics.

```
#pragma float_control( except, off )    // disable exception semantics
#pragma float_control( precise, off )   // disable precise semantics
#pragma fp_contract(on)                 // enable contractions
#pragma fenv_access(off)                // disable fpu environment
sensitivity
```

Note   Exception semantics must be turned-off before turning off "precise" semantics.

Enabling Floating-Point Exception Semantics

Certain exceptional floating-point conditions, such as dividing by zero, can cause the FPU to signal a hardware exception. Floating-point exceptions are disabled by default. Floating-point exceptions are enabled by modifying the FPU control word with the **_controlfp** function. For example, the following code enables the divide-by-zero floating-point exception:

```
_clearfp(); // always call _clearfp before
enabling/unmasking a FPU exception
_controlfp( _EM_ZERODIVIDE, _MCW_EM );
```

When the divide-by-zero exception is enabled, any division operation with a denominator equal to zero will cause a FPU exception to be signaled.

To restore the FPU control word to the default mode,
`call _controlfp(_CW_DEFAULT, ~0).`

Enabling floating-point exception *semantics* with the `fp:except` flag is not the same as enabling floating-point exceptions. When floating-point exception *semantics* are enabled, the compiler must account for the possibility that any floating-point operation might throw an exception. Because the FPU is a separate processor unit, instructions executing on the FPU can be performed concurrently with instructions on other units.

When a floating-point exception is enabled, the FPU will halt execution of the offending instruction and then signal an exceptional condition by setting the FPU status word. When the CPU reaches the next floating-point instruction, it first checks for any pending FPU exceptions. If there is a pending exception, the processor traps it by calling an exception handler provided by the Operating System. This means that when a floating-point operation encounters an exceptional condition, the corresponding exception won't be detected until the next floating-point operation is executed. For example, the following code traps a divide-by-zero exception:

```
double a, b, c;
. . .

...unmasking of FPU exceptions omitted...
__try
{
   b/c; // assume c==0.0
   printf("This line shouldn't be reached when c==0.0\n");
   c = 2.0*b;
}
__except( EXCEPTION_EXECUTE_HANDLER )
{
   printf("SEH Exception Detected\n");
}
```

If a divide-by-zero condition occurs in the expression `a=b/c`, the FPU won't trap/raise the exception until the next floating-point operation in the expression `2.0*b`. This results in the following output:

```
This line shouldn't be reached when c==0.0
SEH Exception Detected
```

The **printf** corresponding to the first line of the output should not have been reached; it was reached because the floating-point exception caused by the expression `b/c` wasn't raised until execution reached `2.0*b`. To raise the exception just after executing `b/c`, the compiler must introduce a "wait" instruction:

```
. . .
__try
{
   b/c; // assume this expression will cause a "divide-by-zero" exception
```

```
    __asm fwait;     printf("This line shouldn't be reached when c==0.0\n");
    c = 2.0*b;
} . . .
```

This "wait" instruction forces the processor to synchronize with the state of the FPU and handle any pending exceptions. The compiler will only generate these "wait" instructions when floating-point semantics are enabled. When these semantics are disabled, as there are by default, programs may encounter synchronicity errors, similar to the one above, when using floating-point exceptions.

When floating-point semantics are enabled, the compiler will not only introduce "wait" instructions, it will also prevent the compiler from illegally optimizing floating-point code in the presence of possible exceptions. This includes any transformations that alter the points at which exceptions are thrown. Because of these factors, enabling floating-point semantics may considerably reduce the efficiency of the generated machine code thus degrading an application's performance.

Floating-point exception semantics are enabled by default under the fp:strict mode. To enable these semantics in the fp:precise mode, add the `-fp:except` switch to the compiler command-line. Floating-point exception semantics can also be enabled and disabled on a function-by-function basis using the **float_control** pragma.

Floating-Point Exceptions as C++ Exceptions

As with all hardware exceptions, floating-point exceptions do not intrinsically cause a C++ exception, but instead trigger a structured exception. To map floating-point structured exceptions to C++ exceptions, users can introduce a custom SEH exception translator. First, introduce a C++ exception corresponding to each floating-point exception:

```
class float_exception : public std::exception {};

class fe_denormal_operand : public float_exception {};
class fe_divide_by_zero : public float_exception {};
class fe_inexact_result : public float_exception {};
class fe_invalid_operation : public float_exception {};
class fe_overflow : public float_exception {};
class fe_stack_check : public float_exception {};
class fe_underflow : public float_exception {};
```

Then, introduce a translation function that will detect a floating-point SEH exception and throw the corresponding C++ exception. To use this function, set the structured-exception handler translator for the current process thread with the `_set_se_translator(…)` function from the runtime library.

```
void se_fe_trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )
{
    switch (u)
    {
    case STATUS_FLOAT_DENORMAL_OPERAND:   throw fe_denormal_operand();
    case STATUS_FLOAT_DIVIDE_BY_ZERO:     throw fe_divide_by_zero();
   etc...
    };
}
. . .
_set_se_translator(se_fe_trans_func);
```

Once this mapping is initialized, floating-point exceptions will behave as though they are C++ exceptions. For example:

```
try
{
    floating-point code that might throw divide-by-zero
   or other floating-point exception
}
catch(fe_divide_by_zero)
{
    cout << "fe_divide_by_zero exception detected" << endl;
}
catch(float_exception)
{
    cout << "float_exception exception detected" << endl;
}
```

Related Books

Methods and Applications of Interval Analysis

Improving Floating-Point Programming

References

[1]Goldberg, David
"What Every Computer Scientist Should Know About Floating-Point Arithmetic"
Computing Surveys, March 1991, pg. 203

About the author

Eric Fleegal works for the Visual C++ group at Microsoft where he is responsible for determining the quality of floating-point code generation. His interests include validated numerical computing and generative programming. His most recent research involves validating floating-point optimizations using interval-analysis.