FPCE Recommendations for Compilers

This appendix gives some recommendations for what compilers should implement to comply with the FPCE technical report. The PowerPC Numerics library provides much of this compliance, but some aspects of the report must be implemented by the compiler. This appendix describes those features that must be implemented in the compiler and recommends how they should be implemented. You should read this appendix if you are a compiler designer, or if you are a programmer and want to know what numeric features to look for in your compiler.

Environmental Access Switch

To allow compilers to better optimize applications without ignoring the floating-point environment altogether, the FPCE technical report defines the following pragma to be used as an environmental access switch:

```
#pragma fenv_access on | off | default
```

The **environmental access switch** specifies whether an application may access the floating-point environment. Access to the floating-point environment must occur as if at run time, whereas optimizations occur at compile time. At compile time, the default (to nearest) rounding mode is in effect and all exception flags are clear (this is the default environment). Without an environmental access switch, the compiler must always assume that every floating-point expression might produce an exception, and therefore the compiler cannot perform some types of optimizations (such as forward and backward code motion) on floating-point expressions.

If the environmental access switch is supported, whenever programmers use any of the environmental control functions (described in Chapter 8, “Environmental Control Functions”), they should first turn on the switch. Where the switch is on, the compiler does not fully optimize floating-point expressions, because it assumes that that part of the application can access the floating-point environment. (Accessing the floating-point environment means setting the rounding direction or reading the status of the exception flags.) Where the switch is off, the compiler can fully optimize any floating-point expression because it assumes that that part of the application does not access the floating-point environment. If the application accesses the floating-point environment when the switch is off, the result is undefined.

If an application uses the default rounding mode and does not access floating-point exception flags, the programmer may turn off the environmental access switch, allowing the application to be fully optimized. If the application contains modules that must access the floating-point environment, the programmer must turn on the environmental access switch in those modules and turn it off in all other modules. In this way, the modules that do not require access can be fully optimized.

The FPCE technical report recommends these programming conventions:

- A function call must not alter its caller's modes, clear its caller's flags, or depend on the state of its caller's flags unless the function is so documented.
- A function call is assumed to require default modes unless its documentation specifically promises otherwise or unless it does not contain floating-point expressions.
- A function call is assumed to have the potential of raising floating-point exceptions unless its documentation specifically promises otherwise or unless it does not contain floating-point expressions.
- At compile time, the default environment is in effect.

These conventions allow the programmer to ignore the floating-point environment altogether if default modes are sufficient for the application or function.

Where supported, the `fenv_access` pragma can occur only outside external declarations. It enables or disables compiler optimizations until another `fenv_access` pragma is encountered or until the end of the module. The default state for `fenv_access` is implementation dependent.

Contraction Operator Switch

To allow programmer control of whether contraction operators are used, the FPCE technical report defines the following pragma:

```
#pragma fp_contract on | off
```

When the `fp_contract` pragma is turned on, the compiler can produce contraction operators in the generated code. For the PowerPC processor, the contraction operators are the multiply-add instructions. These instructions perform a multiplication operation and either an addition or a subtraction operation with at most a single roundoff error. For some input values, the result of a multiply-add instruction is slightly different than if the operations were performed separately. This difference in value might be unacceptable in certain programs. Compilers that support the `fp_contract` pragma allow programmers to disable the generation of multiply-add instructions where necessary.

Where supported, the `fp_contract` pragma can occur only outside external declarations. It enables or disables contraction operators until another `fp_contract` pragma is encountered or until the end of the module. The default state for `fp_contract` is implementation dependent.

Hexadecimal Floating-Point Constants

The FPCE technical report expands the definition of a floating-point constant in C to include hexadecimal floating-point constants. This format makes it easier to represent constants equal to or near arbitrary powers of 2 because they can be represented in hexadecimal instead of having to be converted to decimal.

A hexadecimal floating-point constant has the form

`0x`*hex_digit_seq*`[.`*hex_digit_seq*`]p[+|-]`*binary_exponent*`[`*suffix*`]`

which is interpreted as

$$hex\_digit\_seq.hex\_digit\_seq \times 2^{(+|-)binary\_exponent}$$

hex_digit_seq	A sequence of hexadecimal digits. The first digit sequence must be preceded by the characters `0X` or `0x`. The hexadecimal point and the digit sequence appearing after it are optional.
binary_exponent	A decimal integer representing a power of 2. The exponent may or may not have a sign, but it must be preceded by the character `p`.
suffix	One of the standard C floating-point constant suffixes such as `f` for `float`. All floating-point constants are type `double` unless specified otherwise.

Some examples of hexadecimal floating-point constant expressions are

```
0x1.1111p–2    /* interpreted as 1.1111₁₆ × 2⁻² */
0x256p35f      /* interpreted as 256₁₆ × 2³⁵ */
```

Implementing an Expression Evaluation Method

Though PowerPC Numerics can recommend certain expression evaluation methods, these methods must be implemented by the compiler. As described in Chapter 3, "Expression Evaluation," compilers may or may not support widest-need evaluation. This section describes

- the advantages and disadvantages of supporting and not supporting widest-need evaluation

- some special issues compilers must consider regarding evaluating floating-point constants and initializing floating-point variables
- the FPCE-recommended macros and pragmas that help programmers use the most efficient types possible and determine which expression evaluation method is being used

Expression Evaluation Without Widest Need

The main advantage of using an expression evaluation method without widest-need evaluation is that it is simple to implement. The PowerPC architecture is based on single-precision and double-precision operations, so either single or double is a logical choice for the minimum evaluation format.

Choosing single as the minimum format provides the highest performance for single-precision algorithms yet still allows double and double-double algorithms to be performed with greater precision and range. A single minimum evaluation format, then, allows the best possible performance for all expressions by allowing the semantic type of a simple expression to determine its evaluation format.

Choosing double as the minimum format provides extra precision and range to single-precision operations and conforms to the traditional behavior of the C programming language (traditional C performs all floating-point operations in double precision). Performing all single-precision operations in double precision protects the operations against roundoff errors and against encountering an overflow or underflow in an intermediate value. For example, consider the following expression:

$$\frac{10^{38} \times 10^{20}}{10^{20}}$$

If you perform this expression by hand, you get 10^{38}. If all constants are in single format, the expression produces $+\infty$. The constant 10^{38} is near the end of the range of single format. Multiplying by 10^{20} produces 10^{58}, which is rounded to $+\infty$. Then, $+\infty$ is divided by 10^{20}, and the answer is still $+\infty$.

If the minimum evaluation format is double, the constants 10^{38} and 10^{20} are converted to double format before the result is calculated. The multiplication operation no longer overflows the range of the data type because the double format can easily hold 10^{58}. The value 10^{58} divided by 10^{20} produces 10^{38}, which is then converted back to single format.

Choosing the double-double format provides the greatest available precision to all floating-point operations, protecting double-precision operations as well as single-precision operations from roundoff errors. However, it significantly decreases performance for those expressions that would normally be evaluated in a narrower format. In most cases, the extra precision is not necessary.

Imposing a narrow format allows the best possible performance for narrow-format operations but might produce more roundoff errors in places where the extra precision really is necessary. Using widest-need evaluation for complex expressions in conjunction with a minimum evaluation format minimizes the disadvantages of choosing one minimum evaluation format.

Expression Evaluation With Widest Need

Widest-need evaluation provides some of the advantages of using double-double as the minimum format while eliminating the pitfalls. With widest-need evaluation, if an expression contains a double-double variable, all other variables in that expression will ultimately be converted to double-double format, thus reducing the chance of roundoff error in these expressions. If an expression does not contain a double-double variable, widest-need evaluation allows the expression to be evaluated in the narrowest format possible, allowing the best possible performance for that expression.

Widest-need evaluation can seriously inhibit the common subexpression removal optimization for subexpressions of narrower types. If the type of a subexpression is narrower than the type of its enclosing expression, the format of the enclosing expression is imposed on that subexpression. The subexpression's operands are converted to the wider format. Because the conversion must occur as if at run time, the common subexpression removal optimization is in effect disabled for this subexpression.

Floating-Point Constant Evaluation

When a floating-point constant expression appears in a program, the expression evaluation method determines its evaluation format. When widest-need evaluation is not used, the constant is the wider of the minimum evaluation format and the semantic type of the expression. With widest-need evaluation in effect, the constant is converted to the evaluation format of the complex expression it is part of.

In most cases, floating-point constant expressions must be evaluated as if at run time, although they may actually be evaluated at compile time. At compile time, the default rounding direction is in effect, and no floating-point exceptions may be flagged. (These conditions are known as the default floating-point environment. See Chapter 4, "Environmental Controls," for more information.) However, if evaluation takes place as if at run time, the floating-point environment may affect or be affected by the evaluation. This means that if an expression is unexceptional and the default rounding direction is in effect, the expression can be evaluated at compile time. If the expression is exceptional or the current environment is not in the default state, the expression must be evaluated at run time.

In the following two cases the evaluation always takes place at compile time:

- The constant expression appears within the declaration of a variable explicitly declared to be static:

  ```
  static double x = 0.3 + 0.3;
  ```

- The constant expression appears within the declaration of an aggregate type variable (array, structure, or union):

  ```
  struct {int x = 0; double y = 0.3 + 0.3;} numbers;
  ```

The requirement that floating-point constant expressions be evaluated as if at run time usually inhibits the constant folding optimization, in which values of constants are combined at compile time to produce fewer operations at run time. However, constant folding can occur

- if a floating-point constant expression is required to be evaluated at compile time (that is, if the expression is part of the declaration of either an explicitly declared static variable or an aggregate type)
- if the evaluation of the expression at compile time has exactly the same results as it would if evaluated at run time. This can happen under the following conditions:
  - If an expression evaluates to be nonexceptional at compile time, it would also evaluate to be nonexceptional at run time.
  - If the expression appears in a portion of the program where access to the floating-point environment is disabled, the default environment will be in effect at run time, just as it is at compile time.

The following example illustrates when floating-point constant expressions are evaluated:

```
#pragma fenv_access on
void f(void) {
   float w[] = {0.0 / 0.0};      /* no exception raised */
   static float x = 0.0 / 0.0;   /* no exception raised */
   float y = 0.0 / 0.0;          /* exception raised */

   x = 1.0 / 4.0;                /* exact (no exception raised) */
   y = 1.0 / 3.0;                /* exception raised */
}
#pragma fenv_access off
void g(void) {
   double z;

   z = 0.0 / 0.0;                /* no exception raised */
}
```

In the declaration of the array `w`, a floating-point constant expression contains division by zero. This operation is evaluated at compile time because it appears in the declaration of an aggregate type. Similarly, the division by zero in the declaration of `x` is evaluated at compile time because it is declared static. Neither of these expressions generates an exception, because they occur at compile time, although the compiler should generate a warning message in each case.

The next declaration (of `float y`) also includes the expression 0.0/0.0. This expression is evaluated at run time, and the invalid-operation exception is raised.

The first statement in function `f` assigns to `x` the value of the floating-point constant expression 1.0/4.0. The compiler looks at this expression to determine if it will raise any exceptions. The expression is found to be exact, so the compiler can optimize it.

The second statement of the function `f` assigns to `y` the value of the floating-point constant expression 1.0/3.0. The compiler determines that this expression will raise the inexact exception, so it must be evaluated at run time. The compiler cannot optimize it.

Finally, function `g` assigns to the double variable `z` the value of the floating-point constant expression 0.0/0.0. This statement appears after the `fenv_access` pragma has been turned off. This pragma (described in the section "Environmental Access Switch" on page D-1) signals to the compiler that the default environment will be in effect at run time. Because exceptions are disabled in the default environment, this statement will not raise a run-time exception, and so it may be evaluated at compile time and optimized.

Initializing Floating-Point Objects

A program achieves better performance if it initializes data (including floating-point data) at compile time. The degree to which this is possible depends on the programming language and the compiler options that are supported.

As specified for the C programming language, floating-point constant expressions are generally evaluated as if at run time. This includes floating-point constants that initialize floating-point variables. However a floating-point variable may be initialized at compile time

- if the variable is declared to be static

  ```
  static float x = 0.3;
  ```

- if the variable is part of an aggregate type

  ```
  struct {int x = 0; float y = 0.3;} numbers;
  ```

- if the initializing value is nonexceptional (exact) and is in the format of the variable

  ```
  double y = 0.0;
  float x = 0.0f;
  ```

- if access to the floating-point environment is disabled in the part of the program where the variable is initialized

  ```
  #pragma fenv_acess off
  float x = 0.3;
  ```

For programming languages other than C, the data initialization model may be simpler. For example, in Fortran static initialization is accomplished with the DATA statement (embedded in a BLOCK DATA subprogram for labeled COMMON initialization), and the initializing values may only be constants or parameters. Such initialization is accomplished as if at compile time. Variables not initialized by the DATA statement are considered uninitialized and are assigned values at execution time with executable statements.

Data initialization rules for Pascal compilers are implementation defined and must be fully documented. In MPW Pascal targeting 680x0-based Macintosh computers, for example, a unit requiring initialization of its data declares a public procedure, called at execution time by the host program, that performs the initialization. Apple II Pascal, on the other hand, supports an initialization section within the unit.

Compiler Extensions for Expression Evaluation

The FPCE technical report recommends that compilers implement two macros that help a programmer determine which expression evaluation method is being used and three pragmas that help a programmer use the most efficient data type for functions.

Determining the Expression Evaluation Method

Two macros that characterize the evaluation method for floating-point expressions may be defined in the `float.h` header file. The macro `_MIN_EVAL_FORMAT` tells which numeric data format is used as the minimum evaluation format:

0	`float` (single)
1	`double`
2	`long double` (double-double)

The macro `_WIDEST_NEED_EVAL` specifies if widest-need evaluation is performed:

0	no
1	yes

Widening for Efficiency

In general, programmers want to use the most efficient floating-point data type for the architecture on which their applications will run. If the application is to run on more than one architecture, you cannot guarantee that the most efficient type on one architecture will be the most efficient type for the others. The FPCE technical report recommends three preprocessor pragmas to facilitate running the same application efficiently on different architectures. When these pragmas are turned on, the compiler uses the wider of the architecture's most efficient type and the declared type for any function, parameter, or local variable declared after the pragma.

```
#pragma fp_wide_function_returns    on | off
#pragma fp_wide_function_parameters on | off
#pragma fp_wide_variables           on | off
```

If the first pragma, `fp_wide_function_returns`, is turned on in a module, all of the functions defined below the pragma will have return values in the most efficient data type for the architecture if it is wider than the declared return type. If the following example is compiled for the 680x0 architecture, both functions `ffunc` and `ldfunc` return type `long double`. If compiled for the PowerPC architecture, `ffunc` returns type `double` and `ldfunc` returns type `long double` (because data types may be widened to the most efficient type but not narrowed).

```
#pragma fp_wide_function_returns on
float ffunc (float f) { /* code for ffunc */ }
long double ldfunc (double y) { /* code for ldfunc */ }
```

If the second pragma, `fp_wide_function_parameters`, is turned on in a module, all of the parameters for all of the functions defined below the pragma are converted to the most efficient data type for the architecture if it is wider than the declared types of the parameters. In the following example, the parameters `x` and `y` are both type `double` on the PowerPC architecture and type `long double` on the 680x0 architecture. If an architecture's most efficient type was `float`, the types for both parameters would remain the same (because a parameter's type may be widened to the most efficient type but never narrowed).

```
#pragma fp_wide_function_parameters on
float func(float x, double y) { /* code for func */ }
```

If the third pragma, `fp_wide_variables`, is turned on in a module, all local variables defined below the pragma are converted to the most efficient data type for the architecture if it is wider than the declared types of the variables. In the following example, the variables `z` and `q` are both type `double` on the PowerPC architecture and type `long double` on the 680x0 architecture. If an architecture's most efficient type was `float`, the types for both variables would remain the same (because a variables's type may be widened to the most efficient type but never narrowed).

```
#pragma fp_wide_variables on
float func(float x)
{
   float z;
   double q;

   /* code */
}
```

These pragmas can occur only outside external declarations. Each pragma remains in effect until it is explicitly turned off or until the end of the module. The default state for all three pragmas is off.

If an address or `sizeof` operator is applied to a widened parameter or variable, a compile-time warning is issued. Casts avoid widening in areas where one of these pragmas is turned on.