Language Support for IEEE 754 Floating Point Arithmetic

Joseph D. Darcy

- IEEE 754 is a widespread floating point standard in use for over ten years
- The standard's features allow for clearer, more efficient algorithms to be written

but

- current programming languages do not support the standard
- compiler writers and language designers are typically not aware of floating point issues
- providing a library interface to IEEE 754 features is insufficient
- to be useful, IEEE features must be incorporated into the language directly

IEEE 754 Specific Features

- Special values, all operations can yield an IEEE value
  - ±Infinity to represent overflow and divide by zero
  - NaN (not a number) to represent 0/0, $\infty - \infty$, etc.
  - gradual underflow with subnormal numbers to ease error analysis
- Dynamic Rounding modes
  - useful for interval arithmetic
  - can meaningfully run identical code under different rounding modes, e.g. upper and lower bound of a polynomial and its derivative
  - powerful tool to verify sensitivity at different inputs
- Floating point exceptions and flags
  - exceptions to transfer control
  - flags to have a record of certain events in program execution

Optimization Hazards

- A simple innocuous function

```
int devious(void){
int f, g;
f = (1.0f + 5.96047e-08f) == 1.0f;
g= (-1.0f - 5.96047e-08f) == -1.0f;
return 2*(f^g) + (f*(1-g)) + (1-f)*(1-g);
}
```

constant folding

```
int devious(void){
int f, g;
f = (1.0000001f) == 1.0f;
g= (-1.0000001f) == -1.0f;
return 2*(f^g) + (f*(1-g)) + (1-f)*(1-g);
}
```

more constant folding

```
int devious(void){
int f, g;
f =0;
g= 0;
return 2*(f^g) + (f*(1-g)) + (1-f)*(1-g);
}
```

constant propagation

```
int devious(void){
return 1;
}
```

True purpose of original program

```
fp_rnd getRound(void){
int f, g;
// FLT_EPSILON is the smallest value such that
// 1 + FLT_EPSILON ≠1, but under certain
// rounding modes 1 + FLT_EPSILON == 1

f = (1.0f + FLT_EPSILON) == 1.0f;
g= (-1.0f - FLT_EPSILON) == -1.0f;

// Return rounding mode encoding:
// 0 = to zero
// 1 = to nearest
// 2 = to positive infinity
// 3 = to negative infinity
return (fp_rnd)(2*(f^g) + (f*(1-g)) + (1-f)*(1-g));
}
```

- rounding modes are not in current language semantics
- optimizers can assume the rounding mode is always "round to nearest" and optimize accordingly
- two choices
  - never perform such optimization on floating point expressions
  - add rounding modes to language so that the compiler can perform analysis to inhibit such optimizations when appropriate
- similar argument for sticky flags

Language Support

- New floating point constants (`nan`, `infinity`, `-0`)
- block scoped `round` declarations to change rounding mode
- operator overloading to create user defined numeric types
- new user defined operators
  - rounding mode operators for fine grained control, `a +^ b`
  - quiet comparison operators recommended in standard, `a >=? b`
  - matrix operators
- `enable`/`disable` declarations to control throwing of floating point exceptions

Algorithmic Efficiencies

- Goal: Find the 2 norm of a vector $\sqrt{\sum_{i=1}^{n} v_i^2}$

  ```
  double two_norm(vector v)
  {
  double sum=0.0;
  for(int i=1; i <= v.length; i++)
          sum += v[i]*v[i];
  return sqrt(sum);
  }
  ```
- Problem: Intermediate sums could easily exceed the overflow or underflow threshold of the floating point format
- Algorithm must be robust and work for all cases

2 Norm of a vector (cont.)

- scan entire vector looking for maximum and minimum value and scale accordingly (each element touched twice rather than once) and scaling process is non-trivial
- BLAS routine dnrm2
  - single pass of data, *but*...
  - eight conditional tests in loop to cover all cases (bad for branch prediction, pipelining)
  - calculations on *all* vectors are slowed down by testing for conditions that rarely occur
- IEEE aware code is almost as short and clear as naive case
  - Exception handling is used to make common case run fast

IEEE Vector Norm

```
double two_norm(vector v)
{
double sum=0.0;
int range_exceed = 0;
// Turn on trapping mode to throw exceptions
enable overflow, underflow;
for(int i=1; i <= v.length; i++)
{
try{sum += v[i]*v[i];}

// while keeping mantissa bits, count the number of times overflow and underflow occurs
catch
(overflow e){sum = e.value; range_exceed++;}          //e.value is exponent adjusted value of exceptional operation
(underflow e){sum = e.value; range_exceed--;}
}
//adjust exponent of sum to reflect overflow and underflow
sum = scalb(sum, range_exceed*BIAS_ADJUST);
return sqrt(sum);
}
```

Conclusions

- IEEE features are not supported in current programming languages
- a library interface will not work, the language must incorporate and understand the features being used
- using IEEE 754 features can lead to clearer and faster code for scientific algorithms
- previous work has shown significant speedups are possible for a number of LAPACK routines [James Demmel and Xiaoye Li, “Faster Numerical Algorithms via Exception Handling,” IEEE Transactions on Computers, vol. 43, no 8, August 1994]
- languages and compilers should no longer omit support for IEEE 754 floating point