NAN propagation versus fault trapping in floating point code

By Agner Fog. Technical University of Denmark.

Contents

1 Introduction

The most common ways of detecting numerical errors during floating point calculations are:

1. Exception handling through the trapping of faults
2. Reading the global variable errno
3. Checking end results for infinity (INF) and not-a-number (NAN)

This document will review the advantages and disadvantages of each of these alternatives, with a particular focus on optimizing program performance. We will also look at compiler optimization options.

2 Fault trapping

Microprocessors with hardware support for floating point calculations have a feature for raising exceptions in the form of traps (software interrupts) in case of numerical errors such as overflow, division by zero, and other illegal operations. Fault trapping can be enabled or disabled by setting a control word. The x86 platform has two control words, the "x87 FPU

Control Word" for the old x87 style instructions and the MXCSR register for SSE and AVX instruction sets.

Fault trapping has the following advantages:

- It is possible to detect an error in a try-catch block.
- A debugger can show exactly where the error occurred.
- It is possible to get diagnostic information because the values of all variables at the time of error are available.
- It is possible to design the software so that it can recover from an error. For example, if a piece of code involving multiplications and divisions causes overflow then you can redo the calculation using logarithms.

The disadvantages of fault trapping are:

- Fault trapping is complicated and time consuming. It will slow down program execution if it happens often.
- A trap without a try-catch block will cause the program to crash with an annoying error message that is difficult to understand for the end user.
- Some optimizations are not possible when fault trapping is enabled.
- The compiler cannot optimize variables across the boundaries of a try-catch block.
- The compiler may not be able to vectorize code that contains branches when fault trapping is enabled.
- The code may behave differently when vector instructions are used. Multiple faults in the same vector instruction generates only one trap. See page 6.
- A CPU with out-of-order processing must use speculative execution for all operations that occur after any operation that might generate a trap, even if the later operations are independent of the operation that might generate a trap. It must roll back these operations in the case that a trap is detected. The necessary bookkeeping for speculative execution requires extra hardware resources.

3 Using errno

The 'errno' pseudo-variable contains a code number indicating the type of the last error. This includes all types of errors, for example file errors, and also floating point errors. Originally, errno was a global variable. In modern compilers, it is implemented as a macro in order to make it thread-safe.

Advantages:

- It is simple to use.

Disadvantages:

- It can indicate only a single error.
- It does not tell where the error occurred.
- It contains no details, only a few error codes are available for numerical errors.
- The implementation as a thread-safe macro makes it somewhat less efficient.
- It relies mostly on fault trapping. This implies the same disadvantages as listed above for fault trapping.

4 Propagation of INF and NAN

The standard representation of floating point numbers includes codes for infinity (INF) and not-a-number (NAN). These codes are used for representing invalid results.

Infinity is coded in the following way. All the bits in the exponent field of the floating point representation are set to 1 and the mantissa bits are set to 0. The sign bit indicates the sign of infinity. A NAN is coded in the same way as infinity, but with at least one mantissa bit set to 1. There are two kinds of NANs. A *quiet* NAN has the most significant bit of the mantissa bit set to 1. A *signaling* NAN has this bit cleared and at least one other mantissa bit set. Propagated NANs are always quiet NANs.[1]

The result of a calculation will be INF in the following cases:

- Overflow. The result is bigger than the maximum possible value for the given precision.
- Division by zero when both operands have the same sign.

The result of a calculation will be -INF in the following cases:

- Negative overflow. The result is less than the negative of the maximum possible value for the given precision.
- Division by zero when the two operands have opposite sign.

The result will be zero in case of underflow.

The result will be NAN in the following cases:

- 0/0
- INF-INF
- INF/INF
- 0*INF
- An argument of a function is out of range, e.g. sqrt(-1) or log(-1).

The output of an operation will be INF or NAN if an input is INF or NAN, with a few exceptions listed below. This is useful because an INF or NAN will propagate to the end result if an error occurs in a series of calculations. We don't have to check for errors after each step in a series of calculations if we can be sure that any error will show up in the end result as INF or NAN.

The advantages of relying on the propagation of INF and NAN are:

- The performance is excellent because no resources are spent on error checking, except for the final result. Current hardware does not take extra time to process INF and NAN inputs.
- The code can be vectorized without any change in the behavior (see p. 6).
- A NAN can contain extra information, called a payload, which will propagate to the end result in most cases. The payload can be used for information about the error. For example, library functions can return an error code in the payload of a NAN result.

The disadvantages are:

- An INF result will be converted to a NAN in cases like INF-INF, INF/INF, and 0*INF.
- INF and NAN results are lost when converting to integer or boolean variables.
- All compare operations involving a NAN will be evaluated as false. The programmer needs to take this into account and decide which way a branch should go in case of a NAN input. See page 5.
- When two NANs with different payloads are combined, only one of the payloads is propagated.
- Some operations fail to propagate NAN inputs. These are listed below.

5 Operations that fail to propagate INF and NAN inputs

5.1 Dividing by infinity

Dividing a finite number by INF gives zero.

5.2 Min and max functions

The IEEE 754-2008 standard defines the functions minNum and maxNum giving the minimum and maximum of two inputs, respectively. These functions do not give a NAN output if one of the inputs is NAN and the other is not a NAN.[1]

A forthcoming revision of the IEEE 754 standard defines two additional functions, named minimum and maximum, that do the same but with propagation of NAN inputs.[2]

The actual implementation of min and max in many compilers differs from both of these standards. A common way of defining min and max is:

min(a, b) = a < b ? a : b, max(a, b) = a > b ? a : b.

As all comparisons involving a NAN return false, we have:

min(NAN,1) = 1, min(1, NAN) = NAN, max(NAN,1) = 1, max(1, NAN) = NAN.

The min and max instructions in the SSE and later x86 instruction set extensions are returning the second operand if one of the operands is NAN, in accordance with these definitions.

As no hardware implementations of min and max with guaranteed propagation of NAN input are available yet, you have to implement these in software.

5.3 Power functions

The pow function fails to propagate a NAN in the special cases pow(NAN,0) = 1 and pow(1,NAN) = 1. You may avoid this problem by replacing pow(x,y) by powr(x,y) when x >= 0, or by pown(x,y) when y is an integer. The three functions differ in the following ways, according to the IEEE 754-2008 standard.[1]

x,y	**pow(x,y)**	**powr(x,y)**	**pown(x,y)**
0, 0	1	NAN	1
NAN, 0	1	NAN	1
1, NAN	1	NAN	not possible
negative, integer	x^y	NAN	x^y
negative, non-int	NAN	NAN	not possible
INF, 0	1	NAN	1
1, INF	1	NAN	not possible

5.4 Other functions

exp(-INF) = 0.
atan(± INF) = ± $\pi/2$. atan2(± INF, ± INF) = ± $\pi/4$.
tanh(± INF) = ± 1.
hypot(NAN, INF) = hypot(INF, NAN) = INF.
compound(INF,0) = compound(NAN,0) = 1

6 Compare operations involving NAN

All comparisons with a NAN input will be evaluated as 'unordered'. The operators `>`, `>=`, `==`, `<`, `<=` will all evaluate as false if one or both operands are NAN. The `!=` operator evaluates as true if one or both operands are NAN. Comparison of a NAN with itself will be false:

```
float a;
if (a == a) {
   // false if a is NAN. True in all other cases
}

if (a != a) {
   // goes here only if a is NAN
}

if (!(a == a)) {
   // goes here only if a is NAN
}
```

You may want to decide which way a branch should go in case of a NAN input. If you want a comparison to be evaluated as true in case of NAN inputs, you may negate the opposite condition. This is illustrated in the following two examples.

```
float a, b;
if (a > b) {
   doIfBigger();
}
else {
   doIfLessThanOrEqual();    // goes here if A or B is NAN
}
```

The next example will do the same, except for NAN inputs:

```
float a, b;
if (!(a <= b)) {
   doIfBigger();             // goes here if A or B is NAN
}
else {
   doIfLessThanOrEqual();
}
```

There is no performance cost to negating a condition because most modern processors have hardware instructions for all cases of comparisons including negated comparisons and 'unordered' comparisons which evaluate as true for NAN inputs.

In addition to the 'ordered' and 'unordered' versions of all compare instructions, there are also 'signaling' and 'quiet' versions of the compare instructions. The signaling versions will generate a trap if an input is NAN, while the quiet versions will behave as explained above.[1] The signaling versions are needed only if the code has no branch that adequately handles the NAN cases.

7 Diagnostic information in NAN payloads

The single precision floating point format has 22 payload bit in NANs, double precision NANs have 51 bits of payload. The payload bits are left-justified so that the most significant 22 bits of the payload are preserved when a NAN is converted from double precision to single precision (in binary format).

The payload is usually preserved when a NAN is propagated through a series of calculations. This makes it possible to propagate diagnostic information about an error to the end result.

Current microprocessors make a zero payload when a NAN is generated by hardware, for example in operations such as 0/0. Non-zero payloads can be generated by software. This may be useful for function libraries that put an error code into the NAN result when parameters are out of range.

NAN payloads are also useful for uninitialized variables. This makes it possible to detect variables that have been used without initialization. You can use a signaling NAN if you want a trap when the variable is loaded, or a quiet NAN if you just want the NAN to propagate.

Unfortunately, the propagation of NAN payloads is not fully standardized. When two NANs with different payloads are combined, the result will be one of the two inputs, but the standard does not specify which one. This is discussed below on page 7.

NAN payloads are also used for non-numeric information in a technique called NAN-boxing. Programming languages with weak typing, such as JavaScript, are sometimes storing text strings and other non-numeric variables as NANs where the payload is a pointer or index to the string, etc.

8 Vector code and SIMD instructions

Many modern microprocessors have vector registers and vector instructions that make it possible to operate on multiple data elements in a single operation, using the so called SIMD principle (Single Instruction Multiple Data). The length of the vector registers has grown exponentially in recent years for the x86 instruction set extensions: MMX = 64 bits, SSE = 128 bits, AVX = 256 bits, AVX512 = 512 bits.

The use of vector instructions can often speed up the code quite a lot, but it may cause problems for code that relies on fault trapping. Consider the following example:

```
float a[32], b[32], c[32];
...
for (int i = 0; i < 32; i++) {
   if (a[i] > 0) {
      a[i] = b[i] + 1;
   }
   else {
      a[i] = b[i] / c[i];
   }
   ...
}
```

An good optimizing compiler may vectorize this loop by loading sixteen consecutive elements of each array into each one 512-bit vector register so that it only has to run the loop body two times. It will compute both of the branches `b[i] + 1` and `b[i] / c[i]` and then merge the two result vectors depending on each of the boolean values of `a[i] > 0`. However, it may happen, for example, that `c[15]` is zero while `a[15] > 0`. This would give a false trap for division by zero in a not-taken branch if fault trapping is enabled. The compiler will fail to vectorize the code for this reason if fault trapping is enabled. Alternatively, the compiler will have to execute each branch conditionally, but I have not seen any compiler capable of doing so.

Even if the code contains no branches, we may still have inconsistent behavior. Assume that we remove the first branch in the above example so that we have the unconditional division `a[i] = b[i] / c[i];`. Without vectorization, we have a trap in the sixteenth iteration of the loop, but with vectorization we have a trap in the first iteration of the loop. If the loop has any side effects, for example writing something to a file, then the two cases with and without vectorization will generate different results.

There can be further complications if more than one element in a vector register are causing faults. We will get only one trap if there are multiple faults in the same vector instruction, even if these faults are of different types.

A code that can unravel all these complications and identify the individual faults will be quite complicated if it relies on fault trapping. It will be much simpler to rely on INF/NAN propagation. We will simply get an INF or NAN in any element of the result that has a fault. These faults will show as 'INF' or 'NAN' if the results are printed out. We will get exactly the same results regardless of whether the code is vectorized or not, and we will get the same results on different microprocessors with different vector register lengths.

The advantage of INF/NAN propagation requires, of course, that these values are propagated to the end result. The code must be analyzed to see if there are any instructions or functions that may fail to propagate INF and NAN values, as explained above.

9 Does the IEEE 754 floating point standard need revision?

The IEEE 754 floating point standard specifies that when two NANs with different payloads are combined, the result will be one of these two, but it is not specified which one.[1]

An expression like NAN1 + NAN2 will most often give the result NAN1, but in some cases it gives NAN2. This means that the same program may give different results in different cases depending on the microprocessor and the compiler. Even small changes in the compiler optimization options may change the result because the compiler is allowed to swap the operands. It would be desirable to have a standard that guarantees consistent and predictable results.

I have discussed this issue with the working group behind the IEEE 754 floating point standard. A possible solution that would guarantee consistent and predictable results would be to propagate the largest of the two payloads. But the working group decided not to make any recommendation on the matter in the forthcoming revision of the standard (2018) because NAN payload propagation is seldom used today and it is difficult to predict future needs.[4]

The standard is also unclear about what happens to the NAN payload when converting to a different precision. Experiments on various brands of microprocessors show that the NAN payload is handled in the same way as the mantissa bits of normal floating point numbers, where the most significant bits are preserved when the precision is converted. (This applies to all binary floating point formats, while the less common decimal formats treat the NAN payload, as well as the mantissa, as integers where the least significant bits are preserved.)[4]

Programmers need to be aware of this when NAN payloads are used with floating point numbers of different precisions. The NAN payload is shifted left 29 places when converting from float to double, and shifted right 29 places when converted from double to float. It may be useful to interpret the NAN payload as a binary fraction, a left-justified integer, a bit-reversed integer, or a string of bits in order to get a definition that is independent of the precision, but the standard does not recommend any solution to this problem. The forthcoming revision of the IEEE 754 standard (2018) defines functions for getting and setting the payloads as integers without regard for consistency across different precisions.[2]

It would be useful to reserve different ranges of payload values for different purposes in order to avoid conflicts or misinterpretation, but it is probably premature to define such ranges because NAN payloads are rarely used today.

Known implementations of NAN boxing are using only the lower 35 bits of double precision signaling NANs. Thus, we may distinguish error codes from NAN boxes by setting at least one of the upper 16 bits of the payload in error codes.

The rule that all comparisons involving a NAN must evaluate as false has many strange consequences. It may cause inconsistent behavior or infinite loops. A further consequence of this rule is that the necessary number of compare instructions that a microprocessor must support is doubled. It often happens during the compilation of a program that a condition is inversed, but the inverse of "a == b" is not "a != b" but "a != b or unordered". (see the examples above on page 5). This principle is implemented in hardware and software at least since 1985 and it cannot be changed now.[3]

10 The effects of compiler floating point optimization options

Strict adherence to the IEEE 754 floating point standard and language standards prevents compilers from optimizing floating point code in many cases. The best compilers have several different optimization options that allow the software developer to sacrifice precision or standard conformance in order to improve calculation speed for floating point code. Some of these options and their consequences are listed below. This list relies on the Gnu compiler which has more detailed and well-defined optimization options than most other compilers.[5]

10.1 Options that effect precision

-ffp-contract=fast	Allow fused multiply-and-add (default if -mfma or -mavx512f). This increases precision and improves speed.
-freciprocal-math	Replace division by a constant with multiplication by the reciprocal. x/10 = x*0.1. This improves speed and has only negligible effect on precision.
-fassociative-math	(a+b)+c = a+(b+c). This can have severe consequences for precision. Example: float a = 1.E10, b = (1+a)-a; This gives 0 with -fno-associative-math and 1 with -fassociative-math. The programmer may prefer to use parentheses to indicate which order of calculation is desired rather than using this option.

10.2 Options that affect overflow

-ffinite-math-only	Optimizations assuming that overflow does not occur.
-fassociative-math	It is possible that (a+b)+c overflows while a+(b+c) does not, or vice versa.
-ffp-contract=fast	It is possible that a*b+c overflows, while a fused multiply-and-add does not.

10.3 Options that affect fault trapping

-fno-trapping-math	x = 0.0/0.0 is reduced to x=NAN, giving no trap.
-ffinite-math-only	Optimizations assuming that overflow does not occur.
-fno-signaling-nans	Assume that signaling NANs do not occur (default).

10.4 Options that affect errno

-fno-math-errno	Don't make a trap that sets errno for sqrt etc.

10.5 Options that affect NAN generation and propagation

-ffinite-math-only	Assume that INF and NAN do not occur. For example: x-x = 0. This will be wrong if x = INF because INF-INF = NAN.
-fno-signed-zeros	x*0 = 0, also requires -ffinite-math-only

10.6 Options that affect signed zero

-fno-signed-zeros	x+0 = x. (will give -0. rather than 0. when x = -0.)

10.7 Options that affect vectorization

-O3	Required for automatic vectorization.
-fno-trapping-math	Necessary for vectorization of code containing branches.
-fno-math-errno	Necessary for vectorization of code containing sqrt.

10.8 Combined options

-funsafe-math-optimizations	Enables -fno-signed-zeros, -fno-trapping-math, -fassociative-math and -freciprocal-math.
-ffast-math	Enables -funsafe-math-optimizations, -fno-math-errno, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans, etc.

11 Detecting integer overflow

The x86 platform allows trapping of integer division by zero, but it cannot trap overflow in integer addition, subtraction, and multiplication. Integer variables have no codes for INF and NAN.

It is particularly difficult to detect overflow in signed integers because the C standard specifies signed integer overflow as 'undefined behavior' which means that the compiler can assume that overflow does not happen, and some compilers will even optimize away an explicit overflow check for this reason.

The following methods may be used in order to protect a program against integer overflow:

- It is too late to check for signed integer overflow after it has occurred. Check for overflow before it occurs:
  ```
  #include <limits.h>
  int a, b, sum;
  if (b > 0 && a > INT_MAX - b)   // a + b will overflow
  if (b < 0 && a < INT_MIN - b)   // a + b will underflow
  sum = a + b;
  ```

- Use unsigned integers. These are guaranteed to wrap around in case of overflow:
  ```
  unsigned int a, b, sum;
  sum = a + b;
  if (sum < a)  // overflow
  ```

- Use 64-bit integers if 32-bit integers might overflow.

- The SSE2 instruction set supports addition and subtraction of 16-bit signed and unsigned integers with saturation. The result will be the maximum in case of overflow and the minimum in case of underflow.

- Use floating point variables for integer numbers. This will prevent overflow in most cases, but there is a risk of loss of precision, and program execution may be slower.

- Some compilers support integer operations with overflow check, see e.g. https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

12 Conclusion

It is convenient and efficient to use INF/NAN propagation rather than fault trapping for detecting floating point errors. This is especially useful if the code can benefit from vectorization.

The programmer needs to be aware of certain problems when relying on the propagation of INF and NAN:

- There are certain situations and certain functions that fail to propagate INF and NAN.

- Floating point compare instructions will evaluate as false when an input is NAN. It is possible to manipulate branch instructions to go the way you want in case of NAN input.

- It is useful to put error codes into NAN payloads, especially in library functions. This payload may be lost when two NANs with different payloads are combined.

13 Notes

1. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008.

2. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-201x (expected 2018).

3. IEEE Standard for Binary Floating-Point Arithmetic. The Institute of Electrical and Electronics Engineers, 1985.

4. Agner Fog. NaN payload propagation - unresolved issues. 2018. http://754r.ucbtest.org/background/nan-propagation.pdf

5. Options That Control Optimization. Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

14 Copyright