Floating-point Bugs in Embedded GNU C Library

Soonho Kong, Sicun Gao, and Edmund M. Clarke

Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. We report serious bugs in floating-point computations for evaluating elementary functions in the Embedded GNU C Library. For instance, the sine function can return values larger than 10^{53} in certain rounding modes. Further investigation also exposed faulty implementations in the most recent version of the library, which seemingly fixed some bugs, but only by discarding user-specified rounding-mode requirements. We discuss our experience in how these bugs were spotted and how they affected the implementation process of our SMT solver dReal.

1 Introduction

We have found floating-point bugs in Linux systems using Embedded GLIBC (EGLIBC) version 2.16 or older. EGLIBC is a variant of the GNU C Library (GLIBC) which is used as the default implementation in many distributions including Debian, Ubuntu, and their variants.

The following C program computes the value of $\sin(-2.437592)$ in double-precision after setting the rounding direction to upward ($+\infty$).

```c
#include <math.h>
#include <fenv.h>
#include <stdio.h>

int main() {
    double x = -2.437592;
    fesetround(FE_UPWARD);
    printf("sin(%f)=%f\n", x, sin(x));
    return 0;
}
```

The IEEE754 standard [3] does not specify correct rounding methods on elementary functions such as the exponential, logarithm, and trigonometric functions. Programmers and engineers usually expect the program to print out an approximated value around $\sin(-2.437592) \simeq -0.64727239229$ with an "acceptable" amount of error. However, they all should agree that the result be in the range between -1.0 and 1.0, even in the worst case.

However, a surprising result appears if we compile and execute the program in a machine running Ubuntu 12.04 LTS (or any system with EGLIBC-2.15).

The output is greater than 10^{53} and it should not be a return value of sine function in any sense.

```
$ gcc exp_bug.c -lm && ./a.out
sin(-2.437592)=191561981424936943059347927032148030287313979209416704.00
```

Here is another C program computing cosh(3.113408) with directed rounding toward $+\infty$. This example is more interesting because it shows different results on Intel and AMD machines, and both of the results have serious problems.

```
#include <math.h>
#include <fenv.h>
#include <stdio.h>

int main() {
    double x = 3.113408;
    fesetround(FE_UPWARD);
    printf("cosh(%f) = %f\n", x, cosh(x));
    return 0;
}
```

In a machine with Intel Core i7 CPU, the program outputs *inf* while a machine with AMD Opteron processor produces -160.191709.

```
[INTEL] $ gcc cosh_bug.c -lm && ./a.out
cosh(3.113408) = inf
[AMD]   $ gcc cosh_bug.c -lm && ./a.out
cosh(3.113408) = -160.191709
```

Note that $\cosh(3.113408) \simeq 11.2710174432$ and both results *inf* and -160.191709 are simply wrong. Moreover, each of the wrong results has significant implications:

– Intel (*inf*): It has a contagious effect in subsequent computations. *inf* is a special value in the IEEE754 standard which indicates an overflow in a computation. If one of subexpressions is evaluated to *inf*, then in general the main expression also becomes infinity ($+\infty$ or $-\infty$) or NaN (Not a Number).
– AMD (-160.191709): Mathematically, $\cosh(x)$ is greater than or equal to 1 for all $x \in \mathbb{R}$. As a result, programmers and engineers write algorithms based on the invariant $\forall x.\ 1 \leq \cosh(x)$. This result, -160.191709, breaks the invariant and could cause an unexpected behavior.

2 Floating-point Bugs in EGLIBC (Ver. 2.16 or Older)

We have tested the implementations of the following math functions in C standard library:

Table 1. Experiment Setup

Function	Domain	Range	Function	Domain	Range
sin	$[-10^{306}, 10^{306}]$	$[-1.0, 1.0]$	acos	$[-1.0, 1.0]$	$[-\infty, +\infty]$
cos	$[-10^{306}, 10^{306}]$	$[-1.0, 1.0]$	asin	$[-1.0, 1.0]$	$[-\infty, +\infty]$
tan	$[-10^{306}, 10^{306}]$	$[-\infty, +\infty]$	atan	$[-1.0, 1.0]$	$[-\infty, +\infty]$
cosh	$[-500, 500]$	$[1.0, +\infty]$	exp	$[-100, 100]$	$[0.0, +\infty]$
sinh	$[-500, 500]$	$[-\infty, +\infty]$	log	$[10^{-306}, 10^{306}]$	$[-\infty, +\infty]$
tanh	$[-100, 100]$	$[-1.0, 1.0]$	log10	$[10^{-306}, 10^{306}]$	$[-\infty, +\infty]$
sqrt	$[0.0, 10^{306}]$	$[0.0, +\infty]$			

`sin`, `cos`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `exp`, `log`, `log10`, `sqrt`.

For each function f, we take $100,000$ random numbers from a subset of function f's domain. Table 1 shows each function's sampling domain and range. We pick the sampling domain carefully so that the result of the computations can be fit in a double-precision variable. We consider the four rounding modes supported by C99 standard [4]:

$$\cdot \text{ (nearest)}, \ \rightarrow\!0 \text{ (toward zero)}, \ \uparrow \text{ (toward } +\infty), \ \downarrow \text{ (toward } -\infty).$$

For each sample x and for each rounding mode rnd, we compute two values $f_C^{\text{rnd}}(x)$ and $f_{\text{MPFR}}^{\text{rnd}}(x)$ where f_C is a function f in C standard library and f_{MPFR} is a function f in the GNU MPFR library. MPFR supports arbitrary-precision floating-point computation and we use it as a reference implementation to have a comparison. The correctness of MPFR is, of course, another issue and we do not discuss it here. In the experiments, we use 256-bit precision for MPFR.

We have the following expectations for the two values:

– Consistency: The difference between $f_C^{\text{rnd}}(x)$ and $f_{\text{MPFR}}^{\text{rnd}}(x)$ should not be too large. In this experiment, we set the threshold as 2^{20} ULP (Unit of Least Precision) which is the spacing between two adjacent floating-point numbers. Note that IEEE754 double-precision format has 53 bits of precision and 2^{20} ULP implies that it loses 20-bit precision out of 53. If $|f_C^{\text{rnd}}(x) - f_{\text{MPFR}}^{\text{rnd}}(x)| > 2^{20}\text{ULP}$, we label the case as "inconsistent".
– Correctness: The value of $f_C^{\text{rnd}}(x)$ should be in the range of the mathematical function f. For instance, $\sin_C^{\text{rnd}}(x)$ has to be between -1.0 and 1.0 no matter how imprecise it is.

We run the experiments[1] on two machines – one with Intel Core i7-2600 CPU (8-core, 3.40GHz) and another with AMD Opteron Processor 6134 (32-core, 2.30GHz). Both of them are running Ubuntu 12.04 LTS in which uses EGLIBC-2.15 for the C standard library implementation. We use MPFR-3.1.1 and g++-4.8.1 C++ compiler in the experiments.

The experimental results are summarized in table 2.

[1] Source code is available at https://github.com/soonhokong/fp-test

Table 2. Experimental results on Intel and AMD machines: $f^{\uparrow}$, $f^{\downarrow}$, and $f^{\rightarrow 0}$ indicate a function f with rounding mode toward $+\infty$, toward $-\infty$, and toward 0 respectively. "Inconsistent" denotes the number of cases in which the difference of two results are larger than 2^{20} ULP (Unit of Least Precision). "Incorrect" denotes the number of cases in which $f_C(x)$ is out of f's mathematical range. "$\pm\infty$" denotes the number of cases in which $f_C(x)$ is either $-\infty$ or $+\infty$.

Function	Intel				AMD			
	Inconsistent	Incorrect	$\pm\infty$	Total	Inconsistent	Incorrect	$\pm\infty$	Total
$\sin^{\uparrow}$	10055	446	0	10501	9853	450	0	10303
$\sin^{\downarrow}$	9619	497	0	10116	10009	450	0	10459
$\sin^{\rightarrow 0}$	10087	436	0	10523	9904	423	0	10327
$\cos^{\uparrow}$	10097	434	0	10531	9880	423	0	10303
$\cos^{\downarrow}$	9815	442	0	10257	9910	461	0	10371
$\cos^{\rightarrow 0}$	9737	444	0	10181	9913	441	0	10354
$\tan^{\uparrow}$	12218	0	0	12218	12452	0	0	12452
$\tan^{\downarrow}$	12387	0	0	12387	12378	0	0	12378
$\tan^{\rightarrow 0}$	12486	0	0	12486	12506	0	0	12506
$\cosh^{\uparrow}$	18768	30139	935	49842	37091	12295	291	49677
$\cosh^{\downarrow}$	49766	0	0	49766	49673	0	0	49673
$\cosh^{\rightarrow 0}$	49713	0	0	49713	49772	0	0	49772
$\sinh^{\uparrow}$	47807	0	0	47807	47451	0	266	47717
$\sinh^{\downarrow}$	47493	0	0	47493	47676	0	0	47676
$\sinh^{\rightarrow 0}$	47911	0	0	47911	48046	0	0	48046
$\tanh^{\uparrow}$	3107	0	0	3107	3242	0	0	3242
$\tanh^{\downarrow}$	3135	0	0	3135	3268	0	0	3268
$\exp^{\uparrow}$	47386	2536	0	49922	37883	11708	124	49715
$\exp^{\downarrow}$	49646	0	0	49646	49978	0	0	49978
$\exp^{\rightarrow 0}$	50022	0	0	50022	49836	0	0	49836

1. The implementations of sin, cos, tan, cosh, sinh, tanh and exp functions have severe problems when they are used with the non-default rounding modes (toward ∞, toward $-\infty$, and toward 0). It is also not rare to have the problematic cases in practice. For example, $\cosh^{\uparrow}$ function gives wrong results almost 50% of cases (49,842 out of 100,000).
2. We have not observed any problem under the default rounding mode (toward nearest representable number). Also the implementations of acos, asin, atan, tanh, log, $\log_{10}$, and sqrt functions pass our tests.

3 Fix in EGLIBC-2.17 and Remaining Problems

In EGLIBC-2.17, they provided a patch for the problem. The following is a part of the new implementation of sine function (IEEE754 double-precision)[2]:

[2] Available at `http://www.eglibc.org/cgi-bin/viewvc.cgi/branches/eglibc-2_17/libc/sysdeps/ieee754/dbl-64/s_sin.c?view=markup`

eglibc-2.17/libc/sysdeps/ieee754/dbl-64/s_sin.c

```
101  __sin(double x){
102          double xx,res,t,cor,y,s,c,sn,ssn,cs,ccs,xn,a,da,db,eps,xn1,xn2;
103  #if 0
104          double w[2];
105  #endif
106          mynumber u,v;
107          int4 k,m,n;
108  #if 0
109          int4 nn;
110  #endif
111          double retval = 0;
112
113          SET_RESTORE_ROUND_53BIT (FE_TONEAREST);
```

We find that the patch does not really fix the problem. At line 113, it resets the rounding mode to “round to nearest” and compute the value of $\sin(x)$ while ignoring the user-specified rounding mode. We found a case in which the value of $\sin_C^{\uparrow}(x)$ is smaller than the value of $\sin^{\cdot}_{\mathrm{MPFR}}(x)$, which violates the semantics of “toward $+\infty$” rounding mode:

$$\begin{aligned}\sin_C^{\uparrow}(-3.93799) &= 0.7148414480838297\underline{66879659928236}\\ \sin^{\cdot}_{\mathrm{MPFR}}(-3.93799) &= 0.714841448083829\underline{771665831705916}\end{aligned}$$

4 Our Experience with the Bugs

We found the bugs in the Embedded GNU C Library while investigating a problem in our SMT solver, dReal [1]. dReal implements a δ-complete decision procedure for nonlinear arithmetic, and should never return an `unsat` answer on a satisfiable formula. However, we obtained wrong answers on several simple satisfiable formulas. During debugging, we concluded that the only place that can go wrong is with interval computation in the ICP (Interval Constraint Propagation) engine, Realpaver [2]. Realpaver uses the C standard library functions to change rounding modes and perform interval computations. For instance, the following is the interval implementation of sinh function in `realpaver-1.0`.

realpaver-1.0/src/rp_interval.c

```
void rp_interval_sinh(rp_interval result, rp_interval i)
{
  RP_ROUND_DOWNWARD();
  rp_binf(result) = sinh(rp_binf(i));
  RP_ROUND_UPWARD();
  rp_bsup(result) = sinh(rp_bsup(i));
}
```

For an input interval $i = [l, u]$, this implementation computes result $= [\sinh^{\downarrow}(l), \sinh^{\uparrow}(u)]$ using math functions in C standard library. Combined with the problem of `eglibc-2.16`, this implementation caused unexpected behaviors when the program is executed in machines running latest Ubuntu/Debian OS. To fix the problem, we change the interval computation in Realpaver using FILIB++ [5], a more reliable interval computation library. However, we remark that unless a floating point library is completely verified, there is always the possibility for obtaining wrong answers. The user should always require dReal to produce a proof of unsatisfiability for `unsat` answers, and validate it using the proof checker [1].

5 Conclusion

We report serious bugs in floating-point computations for evaluating elementary functions in the Embedded GNU C Library. It is not a negligible numerical error but either a significant error (2^{20} ULP) or mathematically incorrect result (i.e. $\sin(x) > 10^{53}$) which can trigger severe problems in the following computations. Moreover, the chances of having these results are not rare at all as we have shown in section 2. The current fix does not mitigate the problem but hides it.

References

1. Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.
2. Laurent Granvilliers and Frédéric Benhamou. Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, March 2006.
3. IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. August 1985.
4. ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.
5. Michael Lerch, German Tischler, Jürgen Wolff Von Gudenberg, Werner Hofschuster, and Walter Krämer. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw.*, 32(2):299–324, June 2006.