Tutorial: the Replication Package of Paper "Detecting and Fixing Precision-Specific Operations for Measuring Floating-Point Errors"

Ran Wang, Yingfei Xiong
Peking University

1 Subjects

We evaluated our approaches on the functions of scientific mathematical libray of the GNU C Library (GLIBC), version 2.19. Each math function in GLIBC contains three versions respectively for float, double, and long double. We take only the double version as subjects, which are listed under the directory `glibc-2.19/sysdeps/ieee754/dbl-64`.

We further using the following criteria to filter the functions. First, we choose only the functions that appear in the manual for GLIBC[1] to filter out helper functions in the source code. Second, we chose a function only when it contains at least one floating-point arithmetic. Third, some functions are written in assembly code, and these functions are excluded. Eventually, our experiment subjects are composed of 48 functions, which are shown in Table 1.

2 Manual Installation

Please note that we also provide a virtual machine where all tools are properly installed. You may skip to the next section if you use our virtual machine (recommended).

In the artifact directory of the GitHub project, you will find a set of files that will be used in the following installation steps. The installation should be performed on Ubuntu version 10.04 64-bit.

2.1 Installing FPDebug

We use FPDebug [1] to tune the precision. To install FPDebug, extract files in `install_fpdebug.tar.gz`, append text in `source.list.txt` to `/etc/apt/sources.list`, and then run the following code.

[1] https://www.gnu.org/software/libc/manual/html_node/Mathematics.html#Mathematics and https://www.gnu.org/software/libc/manual/html_node/Arithmetic.html#Arithmetic

```
sudo apt-get update
sudo apt-get install m4
cd install_fpdebug
cd gmp-5.0.1
./configure
make
sudo make install

cd ../mpfr-3.0.0
./configure
make
sudo make install

cd ../valgrind-3.6.1
./configure
make
sudo make install
```

Finally, you have to modify line 4 in `work/glibcFpdebug.h` to point to the header file `yourpath/install_fpdebug/valgrind-3.6.1/fpdebug/fpdebug.h`, where `yourpath` is the path you extracted the tar.gz file.

2.2 Installing MPFR and GMP

FPDebug uses on a modified version of MPFR, and MPFR depends on GMP. To run FPDebug, we need also install the MPFR and GMP. Because the modified version of MPFR may crash on our subjects, we use the original MPFR in our experiments. Suppose we put `mpfr-3.0.0.tar.gz` and `gmp-5.0.1.tar.gz` in `/usr/src`. Running the following code would install the two packages.

```
cd /usr/src
sudo tar -xzvf mpfr-3.0.0.tar.gz
sudo tar -xzvf gmp-5.0.1.tar.gz
sudo mkdir gmp-build
cd gmp-build
sudo ../gmp-5.0.1/configure --prefix=/usr/src/gmp-build
sudo make
sudo make install
cd ..
sudo mkdir mpfr-build
cd mpfr-build
sudo ../mpfr-3.0.0/configure --prefix=/usr/src/mpfr-build --with-gmp=/usr/src/gmp-build
sudo make
sudo make install
```

To check whether MPFR and GMP is corrected installed, try to run `work/test_mpfr.c`.

```
gcc test_mpfr.c -L/usr/src/mpfr-build/lib -lmpfr -L/usr/src/gmp-build/lib -lgmp -o test_mpfr
./test_mpfr
```

The output should be

```
Sum is 2.718281828459045235360287471352662497757247093699959574966913l
```

If it fails, you can try the following tricky solution.

```
cd /usr/src
cd mpfr-build/lib
sudo mv libmpfr.a libmpfr1.a
cd /usr/src
# another gmp
sudo mkdir gmp-build2
cd gmp-build2
```

```
sudo ../gmp-5.0.1/configure --prefix=/usr/src/gmp-build2
sudo make
sudo make install
cd lib
sudo mv libgmp.a libgmp1.a
# go to the directory where test_mpfr.c is
gcc test_mpfr.c -L/usr/src/mpfr-build/lib -lmpfr1 -L/usr/src/gmp-build2/lib -lgmp1 -o test_mpfr
./test_mpfr
```

2.3 Installing Experimental Scripts

The scripts for implementing our approaches and the experiments are stored in file `work.tar.gz`. These code files can be directly used after decompression. A detailed explanation of the directory structure of the experimental scripts can be found in Figure 1.

2.4 Installing Experimental Subjects

Our experiment code requires the subject files are instrumented first. The instrumentation performs the following three actions. First, it converts the original code into three-address form. Second, it inserts statements for detecting precision-specific operations. Third, it inserts statements for fixing precision-specific operations. The latter two types of actions insert the auxiliary statements for each original statement in the three-address form, where the inserted statements will be used on demand by our controlling script later.

We have provided two instrumented GLIBC packages for our experimental subjects, each using a different strategy for the fixing.

- In package `glibc-2.19-auto.tar.gz`, we fix only the last statement of detected precision-specific operations. Each statement is wrapper with code invoking the fixing mechanism.
- In package `glibc-2.19-manual.tar.gz`, we fix all true positive detections by manually identified ranges, which may be multiple statements.

These two GLIBC packages share the same installation steps, which compile these subjects as static link libraries. Suppose that `glibc-2.19-XXX.tar.gz` and `glibcFpdebug.h` is in `/usr/src`. We can install the subjects using the following commands.

```
cd /usr/src
sudo cp glibcFpdebug.h /usr/include/glibcFpdebug.h
sudo tar -xzvf glibc-2.19-XXX.tar.gz
sudo mkdir glibc-build-XXX
cd glibc-build-XXX
export CFLAGS="-g -O2 -U_FORTIFY_SOURCE -march=native -mtune=native -fno-stack-protector"
sudo ../glibc-2.19-XXX/configure --prefix=/usr/src/glibc-build-XXX --disable-profile --enable-add-ons
sudo make
sudo make install
cd lib
sudo mv libm.a libm1.a
```

Finally, you need to modify the two `glibcmath.h` files in directories `exe-files` and `exe-small` of the experimental code. We need to change the default path

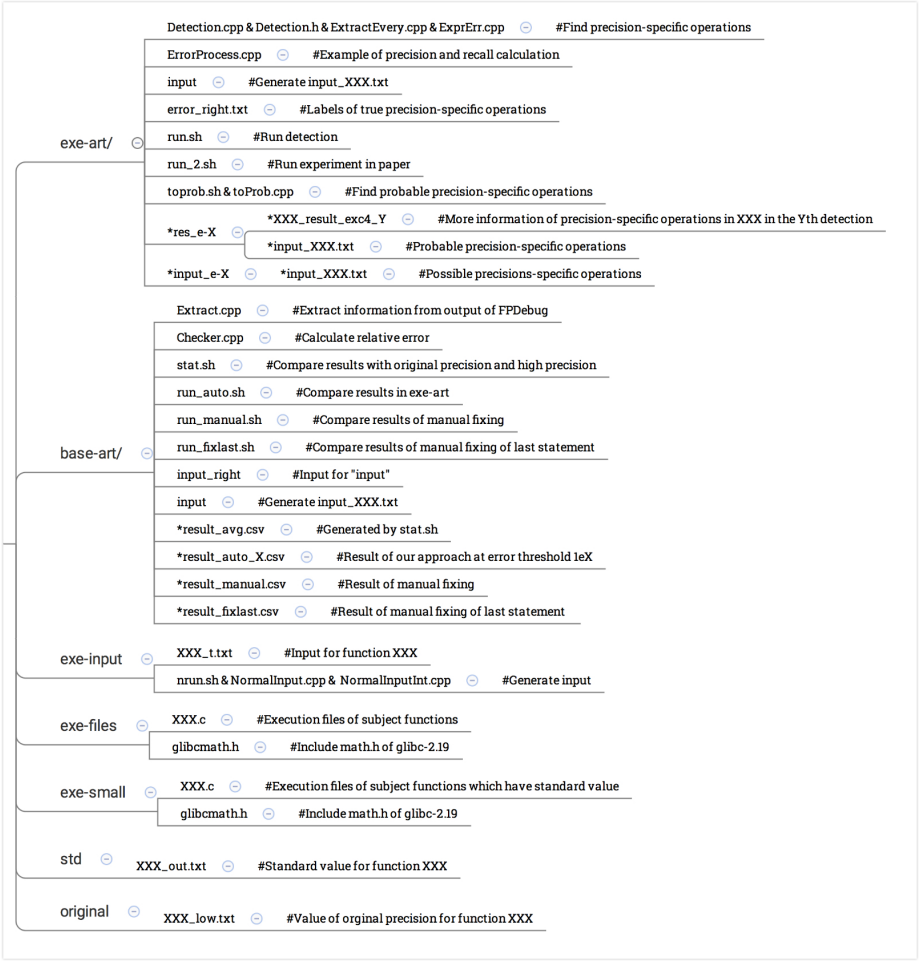

Figure 1: The structure of the experimental scripts

`/usr/src/glibc-build-auto/include/math.h` into the path of `math.h` in any of your installed GLIBC directories. The `math.h` files in the two GLIBC packages are the same.

2.5 Using more subjects

If you want to apply our approach to more subjects, you have to instrument them first. We provide an instrumentation tool to facilitate the process. However, the tool cannot automate all instrumentation and still contains bugs. So manual operations are still needed after using this tool.

If you are interested in applying to more subjects, you can take a look at `run_ind.sh` in `tac.zip` and adapt it for your code. The instrumentation tool is based on LLVM and CLANG. The API version is LLVM 3.6. The current tool may produce uncompilable code and does not support the instrumentation for union data types, which requires manual fix.

3 Installation by Virtual Machine

The second installation method is to use the virtual machine image[2]. The virtual machine is in `ovf` format.

The installation details in the virtual machine are as follows.

1. FPDebug: installed in

   `/home/artifact/work/install_fpdebug/valgrind-3.6.1/fpdebug`

2. MPFR: installed in `/usr/src/mpfr-build`. Library is `libmpfr1`.
3. GMP: installed in `/usr/src/gmp-build2`. Library is `libgmp1`.
4. Experimental scripts: in `/home/artifact/work`. Note that the structure of the experimental scripts is shown in Figure 1.
5. GLIBC subjects: `auto` is installed in `/usr/src/glibc-build-auto`, and its library is `libm1`; `manual` is installed in `/usr/src/glibc-build-manual`, and its library is `libm1`.

4 List of Precision-Specific Operations

File `true-pso.txt` contains the IDs for all precision-specific operations we found in our experiments. This file is located at `/home/artifact/work/true-pso.txt` in the virtual machine image, and is included in the `work.tar.gz` file in the manual installation package.

Each ID is a number that can be used to search the respective statements in the instrumented source. Given an ID, say 47, you can locate the precision-specific operation by searching for code patterns as the follows.

[2] https://1drv.ms/f/s!AkrtmyeJeJbhatpyY6Q4PWhFs8U

```
reducePrecd(&res, 47);
computeErrd("res_tag47_", &res, 47);
reducePrecd(&temp_var_for_tac_23, 47);
computeErrd("temp_var_for_tac_23_tag47_", &temp_var_for_tac_23, 47);
temp_var_for_tac_24 = res + temp_var_for_tac_23;
resumePrecd(&temp_var_for_tac_24, 47);
computeErrd("temp_var_for_tac_24_tag47", &temp_var_for_tac_24, 47);
```

In the above code, `temp_var_for_tac_24 = res + temp_var_for_tac_23` is the precision-specific operation identified. The rest are standard instrumented statements and take the same form for different statements.

5 Running the Experiment

The experiment consists of two steps. The first step is to run detection and fixing approach to find out all precision-specific operations. The second step is to analyze the results. For example, compare the results with standard results, analyze the results of manually fixed.

5.1 Detection and Fixing

In the virtual machine, run the following commands to detect and fix precision-specific operations for our subjects.

```
cd /home/artifact/work/exe-art
./run_2.sh
```

The script repetitively calls `run.sh`, which takes parameters to the installation directories. In the manual installation, you have to manually change `run_2.sh` so that `run.sh` takes the correct parameters. The input parameters for `run.sh` are explained in details when executing `./run.sh -h`. Here we explain the parameters using an example.

```
./run.sh -m /usr/src/mpfr-build -r mpfr1 -g /usr/src/gmp-build2 -p gmp1 -c /usr/src/glibc-build-auto -b m1 -e 1e7
```

The above command means that MPFR is installed in `/usr/src/mpfr-build`, with `libmpfr1.a` under `/usr/src/mpfr-build/lib` (specified by `-r mpfr1`); GMP is installed in `/usr/src/gmp-build2`, with `libgmp1.a` under `/usr/src/gmp-build2/lib` (specified by `-p gmp1`); GLIBC subjects are installed in `/usr/src/glibc-build-auto`, with `libm1.a` under `/usr/src/glibc-build-auto/lib`; the error threshold E is 10^7. These are also default settings for `run.sh`.

After executing the scripts, the results are stored in directories `exe-art/res_e-X` and `exe-art/input_e-X`, where `X` stands for the error threshold 10^X. `input_YYY.txt` in `input_e-X` contains IDs for possibly precision-specific operations with `-1` ends each line. Every line in a particular `input_YYY.txt` is the same. `input_YYY.txt` in `res_e-X` constains IDs for probably precision-specific operations in the same format.

5.2 Analysis

We have four scripts for analyzing detection approach and three fixing approaches.

In the virtual machine, run the following command to evaluate the detection results in `exe-art/res_e-X` for $X = 4, 6, 7, 8, 10$, where X stands for the parameter E of the detection approach.

```
cd /home/artifact/work/exe-art
g++ ErrorProcess.cpp -o ErrorProcess
./ErrorProcess
```

This will print precision and the number of right precision-specific operations and number of wrong precision-specific operations found in the detection. Let u denote the number of right precision-specific operations, the recall can be computed by $\frac{u}{48}$.

`ErrorProcess.cpp` computes precision and recall for probably precision-specific operations. For possibly precision-specific operations, replace `res_e-` with `input_e-` in `ErrorProcess.cpp`.

In the virtual machine, run the following command to evaluate the fully automatically fixed results.

```
cd /home/artifact/work/base-art
./run_auto.sh
```

The result of analyzing fixing is stored in `result_auto_X.csv`. Each row in `result_auto_X.csv` shows results for one subject function. The first column is the name of the function. The 2nd-10th columns are percentage of input that one method beats another. `M` stands for our detection and fixing results, `L` stands for original precision, and `H` stands for high precision. The value of `M>=L` for acos is `1.000000`, which means that our detection and fixing results is *better* than original precision in 100% inputs. *Better* means that the relative error between standard value is smaller. The 11th-13th columns are average relative error between standard value. `mavg` stands for our detection and fixing results, `lavg` stands for original precision, and `havg` stands for high precision.

To evalute the manually fixed GLIBC, run the following command.

```
cd /home/artifact/work/base-art
cp ../exe-small/*.c .
./run_manual.sh
```

Similar as above, the result is stored in `result_manual.csv`.

To evalute the manually fixed GLIBC (last instruction), run the following command.

```
cd /home/artifact/work/base-art
cp ../exe-small/*.c .
./run_fixlast.sh
```

Similar as above, the result is stored in `result_fixlast.csv`.

In the manual installation, again the scripts should be modified so that the paths to the installed tools are properly set. In the script, the central command to be modified is `stat.sh`, which compares the results by some method. Script `stat.sh` takes a set of parameters that are similar to the `run.sh` that we have seen in the previous section. Similarly, a full explanation of the parameters can be obtained by executing `stat.sh -h`.

References

[1] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proc. PLDI*, pages 453–462, 2012.

Selected Function	File
acos	e_asin.c
acosh	wordsize-64/e_acosh.c
asin	e_asin.c
asinh	s_asinh.c
atan2	e_atan2.c
atan	s_atan.c
atanh	e_atanh.c
cbrt	s_cbrt.c
cos	s_sin.c
cosh	wordsize-64/e_cosh.c
erf	s_erf.c
erfc	s_erfc.c
exp10	e_exp10.c
exp2	e_exp2.c
exp	e_exp.c
expm1	s_expm1.c
fma	s_fma.c
fmaf	s_fmaf.c
frexp	wordsize-64/s_frexp.c
gamma	e_gamma.c
hypot	e_hypot.c
ilogb	e_ilogb.c
j0	e_j0.c
j1	e_j1.c
jn	e_jn.c
lgamma	e_lgamma.c
llround	wordsize-64/s_llround.c
log10	wordsize-64/e_log10.c
log1p	s_log1p.c
log2	wordsize-64/e_log2.c
log	e_log.c
logb	wordsize-64/s_logb.c
lround	wordsize-64/s_lround.c
modf	wordsize-64/s_modf.c
pow	e_pow.c
remquo	wordsize-64/s_remquo.c
round	wordsize-64/s_round.c
scalbln	wordsize-64/s_scalbln.c
scalbn	wordsize-64/s_scalbn.c
sin	s_sin.c
sincos	s_sincos.c
sinh	e_sinh.c
tan	s_tan.c
tanh	s_tanh.c
trunc	wordsize-64/s_trunc.c
y0	e_j0.c
y1	e_j1.c
yn	e_jn.c

Table 1: Subjects