CS 6535 - Project Report
Control-flow Instability due to Floating-point Arithmetic

Abhiruchi Karwa
karwa.a@husky.neu.edu

April 18, 2019

Abstract

Floating-point arithmetic is observed to have non-deterministic behaviour. Especially when adhering to standards that make floating-point arithmetic independent of the hardware, software or the combination of both, results obtained should be consistent. Control-flow instabilities are however observed which impairs reliability of the software. The same program when run on the same inputs, produces different results. This is catastrophic when it changes the execution order of the statements in a program. In this project, I present two factors that have made the program behave differently even when no changes were made to the actual code, namely - precision of the floating-point numbers that are involved in the floating-point arithmetic and the level of optimization that the compiler performs with. I also discuss how to generate the critical test cases that help highlight these irregularities in a program.

1 Motivation

Software should be reliable at all times, because unexpected behavior noticed by the user of the software is undesirable. When software failures are caused due to factors unrelated to the actual program(code for the software), they are difficult to diagnose as these factors can not be discovered just by debugging the code. Software unreliability should not be ignored as it may cause the program to behave differently in different environments.
Additionally, floating-point arithmetic is used in various software applications. It is well known that floating-point expressions produce different results in different scenarios owing to factors like lack of associativity, lack of precision or just rounding errors. This is why the IEEE 754 standard was established, to keep these irregularities in check. Unreliability caused due to floating-point arithmetic, is interesting because these same effects are rarely observed due to integer arithmetic and different executions that lead to disparity in evaluation of conditional statements such as *if statements* and loops in a program, could lead to catastrophic errors.

2 Background

In this project, two concepts that are important are - control-flow instability and floating-point arithmetic. One is a phenomenon observed and other causes it - floating-point arithmetic affects the branching decisions made in the code.
As taken from [1], we have the definition as:

Consider a program P, and let q(I,M) denote the value of expression q for program inputs I (that cause q to be reached) and expression evaluation model M.

Definition 2.1 (Control-flow instability) *Let q denote a Boolean-valued expression used as a conditional in program P. Input I is said to cause control-flow instability if there exist two evaluation models M_1 and M_2 such that*

$$q(I,M_1) \neq q(I,M_2).$$

Intuitively, input I is a candidate for causing program P to exhibit different control flows, caused by different Boolean values of the conditional q, for the same input I.

Tangentially, the value of a floating-point number is simply the significand multiplied by the base raised to the power of the exponent, where:

- significand(also referred to as the mantissa, or coefficient) - a signed digit string, of a given length in a given base, where the length of the significand determines the precision to which numbers can be represented.
- exponent(also referred to as the characteristic, or scale) - a signed integer, which modifies the magnitude of the number.

Floating-point numbers are necessary as integers do not represent fractions. So when real numbers need to be represented in a computer program, floating-point numbers are required. However, since computer memory is limited, representing these numbers with infinite precision and exact accuracy is not possible. Whether it be binary or decimal fractions that need to be represented, at some point values have to be rounded off.

Floating-point arithmetic is the arithmetic using formulaic representation of real numbers as an approximation so as to support a trade-off between range and precision. Because floating-point arithmetic can only represent a finite subset of the continuum of real numbers, precision errors are very common on computing applications. Consequently certain properties of real arithmetic, such as associativity of addition, do not always hold for floating-point arithmetic.

In order to have identical results, independent of implementation, given the same input data, the IEEE 754 standard [2] was established to standardize the computer representation for binary floating-point numbers. This standard is followed by almost all modern machines. The standard provides for many closely related formats, differing in only a few details. There are five basic formats and other extended formats. Two widely used formats are:

- Single precision - usually used to represent the "float" type in the C language family (though this is not guaranteed). This is a binary format that occupies 32 bits (4 bytes) and its significand has a precision of 24 bits (about 7 decimal digits).
- Double precision - usually used to represent the "double" type in the C language family (though this is not guaranteed). This is a binary format that occupies 64 bits (8 bytes) and its significand has a precision of 53 bits (about 16 decimal digits).

Moving on to compilation of a program, turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time. The optimization levels in the GNU compiler and their effects on the code can be summarized as follows:

Level of optimization	**Effects**
-O0	No optimization done, runs source code normally.
-O1	Enables optimization for speed and disables some optimization that increases code size
–O2	Enables optimization for code speed which make it perform some basic loop optimization and most common compiler optimization technologies.
-O3	Performs O2 optimization and enables more aggressive loop transformations such as collapsing *if* statements

We can see from the above table how compiler optimization could potentially affect any program. It affects floating-point arithmetic even more when the optimization moves a floating-point instruction past a procedure-call intended to deal with a flag [4], or replace expressions like x/x by 1 (wrong if x is 0 or NaN) and $x - x$ by 0 (wrong if x is NaN) and $0 * x$ and $0/x$ by 0 (wrong if ...).

Now that we know how floating-point numbers with different precisions are stored and how compiler optimization could affect any program, we can delve further into knowing how this causes control-flow instability thus causing software unreliability.

3 Foundational Steps

Given is a program that consists a conditional expression that leads to at least two different resultant control-flows. This conditional expression is a Boolean-valued formula that compares a floating-point valued expression with a floating-point constant. We need to find a test case that will cause the aforementioned conditional expression to compute different Boolean values and flip the control-flow taken further ahead in the

program. Taking help from the algorithm described in [1], my approach to find control-flow instability in programs involved two major steps:

- In the first step using symbolic execution, we can build a propositional logic formula for the path leading from the program entry point to the conditional expression. We have to change the comparison operator in that expression to an equality or in some cases approximate inequality and solve the obtained formula using a constraint solver.
- In the second phase, after obtaining values for the propositional logic using a symbolic execution, I tested them with the code and kept modifying the values until I observed a case of control-flow instability.

This approach is followed in the program submitted as part of this project. The test-case generator program. However, this process is not fully automated yet. The code requires the outputs from the first step, meaning that the symbolic execution and the propositional logic formulation has to be done by the user.

4 Implementation

4.1 Control-flow instability due to changes in precision format

I used the following code in C to test the control-flow instability caused due to the change in precision of the floating-point numbers:

```
int type_of_roots(TYPE a, TYPE b, TYPE c) {
    TYPE discriminant;
    discriminant = b * b - 4 * a * c;
    if (discriminant > 0) {
        printf("Real and distinct roots");
        return 1;
    }
    else if (discriminant == 0) {
        printf("Equal roots");
        return 0;
    }
    else {
        printf("Real and imaginary roots");
        return -1;
    }
}
```

where TYPE is a macro used to change the type of the variables which changes the precision of the floating-point numbers in the program.

Type of precision	Macro used
Single precision	`#define TYPE float`
Double precision	`#define TYPE double`

Following the foundational steps discussed earlier, I found the test cases as follows:

1. I got the following propositional logic formula as : `(assert (and (= (* b b) (* 4 a c)) (not(= a b))))` so as to change the comparison of discriminant into an equality check with *zero*. This led to obtaining the values `0.5, 0.125, 0.0078125`

2. After completing step one, I changed the values obtained earlier and got the following results:

Type of precision	Input test cases	Results obtained
Single precision	`0.5 0.125 0.0078125001`	`Equal roots`
Double precision	`0.5 0.125 0.0078125001`	`Real and imaginary roots`

The test cases specified above make the program give different end results due to change in the precision. I found this particular program interesting as it involves floating-point multiplication. Now, as discussed earlier, double precision has almost double the storage as compared to single precision for storing any floating point number. So, single precision rounds up the value of `discriminant` variable incorrectly whereas double precision doesn't round up those last few digits of the final value and follows a different control-flow.

4.2 Control-flow instability due to changes in level of optimization

I used the following code in C to test the control-flow instability caused due to the change in level of optimization that the compiler works with:

```
float epsilon(float a, float b) {
  float eps = b;
  while (a + eps != a) {
    eps /= (2 * a);
  }
  return eps;
}
int calculate(float input1, float input2) {
  float a = input1;
  float b = epsilon(input1, input2);
  float c = -b;
  float sum = a + b + c;
  if (sum < a) {
      printf("The sum is lesser than the first input");
      return 0;
  }
```

```
    else {
        printf("The sum is greater than or equal to the first input");
        return 1;
    }
}
```

Assuming that the user inputs are passed to the function `calculate`, the following results were obtained after compiling the code with different levels of optimization:

Level of optimization	Input test cases	Results obtained
-O0	`1.0 1.0`	`The sum is greater than or equal to the first input`
-O3	`1.0 1.0`	`The sum is lesser than the first input`

The above code tries to add three numbers depending on the user input such that the latter input gets absorbed into the former. This is ensured by the function `epsilon` in the code, that returns a very small value such that when this value is added to and then subtracted from the first input value, the sum is equal to the first input value. However, when the code is compiled with optimization the loop transformations cause the value returned by `epsilon` to not be very small in comparison to the first input. This doesn't let the sum to be the same as the first input value. I found this program interesting as I wanted some changes in how the actual program is compiled. Thus, displaying a case of control-flow instability.

5 Summary

I have observed and shown the control-flow instabilities due to floating-point arithmetic which have caused the program to have different branches of execution. I have identified two factors that cause control-flow instabilities:

1. Change in precision format of the floating-point numbers in the program
2. Change in the level of optimization used by the compiler

My understanding and reasons for these control-flow instabilities to be observed is also mentioned in the report.
I have also discussed the technique used to find the critical test cases that potentially exhibit control-flow instabilities for a program.
This technique is currently not fully automated and can be used as a part of the black-box test-case generator to find test cases that exhibit control-flow instabilities. Future scope of this project is that instead of manually identifying the critical sub-expression, then converting it into a propositional logic formula and using the values obtained by solving that by symbolic execution to feed to the program, we can make the automated test case generator to do all of this by just taking the program as an input and return the critical test cases.

6 References

1. Gu, Y., Wahl, T., Bayati, M., and Leeser, M.: Behavioral non-portability in scientific numeric computing.

2. Institute of Electrical and Electronics Engineers (IEEE): 754–2008 – IEEE standard for floating-point arithmetic, pp. 1–58. IEEE (2008)

3. Whitehead, N., Fit-Florea, A.: Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs

4. Kahan, W.: Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic

5. A Guide to Vectorization with Intel® C++ Compilers