Stabilizing Numeric Programs against Platform Uncertainties (Extended Abstract)

Yijia Gu and Thomas Wahl

College of Computer and Information Science, Boston, USA

1 Introduction

Floating-point arithmetic (FPA) is a loosely standardized approximation of real arithmetic available on many computers today. The flexibility for floating-point unit designers offered by the widely adopted IEEE 754 FPA standard [1] incurs often underestimated risks for the robustness of embedded software against what we collectively refer to as *platform uncertainties* in this paper. The flexibility includes the freedom for hardware vendors to offer specialized instructions for operations with increased precision (such as *fused multiply-add* [FMA]), and the freedom for compilers to reorder complex expressions more or less at will.

The lack of robustness against platform uncertainties, called *volatility* in this paper, translates in practice to reduced reproducibility of results and, ultimately, non-portability of numeric code. In this Extended Abstract we report on recent results to detect and repair such problems without the need to test the given program against diverse platforms [3]. Our approach computes, for each variable E of interest, a (tight) interval, called the *volatile bound*, such that the value of E is guaranteed to be in that interval, *no matter how the program is compiled*. Large volatile bounds indicate high uncertainty in the computation.

Our technique then proposes ways to reduce platform uncertainty. A naive way is to use compiler flags that enforce strict (deterministic) evaluation, such as `/fp:strict` for Visual Studio `C++`. This unfortunately inhibits optimizations that compilers can apply to harmless (stable) fragments of the code [2]. We present a more fine-grained approach that aims to stabilize only *some* evaluation aspects, of *some* statements S that contribute *most* to the uncertainty in the target expression E. Our approach returns information on what these statements are, and what kinds of uncertainties in S's evaluation are to blame for E's instability. This allows the user to apply fine-grained, local code stabilization.

2 Approach

We use the `C` program fragment shown in Fig. 1 (left) to illustrate our approach. Consider the input

$$\begin{array}{ll} \texttt{r} = \{-10.998046875, -16, -15\}, & \texttt{radiusSq} = 0.015625 \\ \texttt{s} = \{-10.4194345474, -15, -14\}\ . & \end{array} \tag{1}$$

to function `raySphere`. On this input, the program takes different execution paths on an Nvidia GPU and on an Intel CPU. The cause for this unwelcome divergence is a difference in the calculations that propagate to the conditional `if (D > 0)`: the value computed for `D` in the GPU is -0.11892647 (the branch is skipped), while on the CPU it is 0.14415550 (the branch is taken). Depending on what happens in the branch, the behavioral differences of this program on the two platforms can now have unlimited consequences.

```
float dot3(float *a, float *b) {
  return a[0]*b[0] + a[1]*b[1] +
      a[2]*b[2]; }

int raySphere(float *r, float *s,
     float radiusSq) {
  float A, B, C, D;
  A = dot3(r,r);
  B = -2.0 * dot3(s,r);
  C = dot3(s,s) - radiusSq;
  D = B*B - 4*A*C;
  if (D > 0)
    ...; }
```

```
float dot3(float *a, float *b) {
  return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]; }

int raySphere(float *r, float *s, float radiusSq) {
  float A, B, C, D;
  A = dot3(r,r);
  {
    #pragma STDC FP_CONTRACT off
    B = -2.0 * (s[0] * r[0] + s[1] * r[1] + s[2] * r[2]);
  }
  C =((s[0]*s[0] + s[1]*s[1]) + s[2]*s[2]) - radiusSq;
  D = B*B - 4*A*C;
  if (D > 0)
    ...; }
```

Fig. 1: Ray Tracing: original code (left) and locally stabilized version (right)

The numeric uncertainty eventually leading to the decision divergence is due to the presence or absence of FMA hardware instructions on the two processors. FMA is a contraction of expressions of the form `x * y + z` such that the multiplication is in effect performed without rounding. Such expressions come up in Fig. 1 (left) in function `dot3` and in the expression defining `D`.

Running our analysis on this program, it reports a *volatile bound* for `D` of $[-0.252806664, 0.156753913]$. This interval overapproximates the set of values any IEEE 754-compliant compiler/hardware platform can possibly produce, for input (1). Due to the size of this interval, and the looming `D > 0` branch, our analysis warns users that the code may have platform uncertainty problems.

To fix the uncertainty problem, our analysis produces information on the *provenance* of the uncertainty of `D`'s value, i.e., for each preceding statement, a measure of how much its uncertainty contributes to that of `D`. In addition, we output the reason for the uncertainty in each statement, to guide the programmer as to what aspect of the evaluation to fix. This output is shown in Table 1.

Stmt. label	**Provenance** (L, U) **for** `D`	**Reason**
`l1`	(0.000000000, 0.000000000)	null
`l2`	(−0.275680393, 0.000000000)	FMA
`l3`	(−0.146962166, 0.000000000)	order
`l4`	(−0.002806664, 0.031753913)	FMA+order

Table 1: Provenance information for `D` and reason for uncertainty contribution

The left component L ("lower") specifies by how much the left boundary of the volatile bound interval for `D` shifts, due to the numeric uncertainty in the computation at `l`i (analogously for U ["upper"]). That is, a negative value for L indicates that the volatile bound interval expands (to the left). We see that

statement `l2` contributes most to the expansion of `D`'s volatile bound to the left. The uncertainty in `l2` is due to use or non-use of FMA; ordering uncertainty has no effect at all, for input (1).

After disabling FMA *only* for the statement labeled `l2` (details later), our analysis returns a *smaller* volatile bound for `D` of $[-0.002806663, 0.156753913]$. However, the comparison `D > 0` still suffers from decision uncertainty. To further increase the lower bound of `D`, we force left-to-right evaluation order for `l3`. The volatile bound for `D` becomes $[0.125000000, 0.156753913]$, making the comparison stable and consistent with strict evaluation. The locally stabilized version of the Ray Tracing program is shown in Fig. 1 (right).

Accuracy reproducibility experiments. We have also tested our technique on a number of matrix calculation programs; Table 2 shows the results. Column **Input** specifies the input matrix size; $*$ means that the benchmark takes non-matrix inputs. Columns **Volatile Bound** and v_{str} show the volatile bound of the final result for each benchmark and the corresponding value under strict evaluation. If the output is a matrix, we define its volatility to be that of the cell with the largest volatile bound in the matrix, maximized over all test cases; that cell is shown under **Variable**.

Program	Input	Variable	Volatile Bound	v_{str}
sor	$[100 \times 100]$	$G[67][55]$	[0.720786273, 0.720786631]	0.720786452
fft	$[16 \times 2]$	$X[14]$	[0.123653859, 0.123654306]	0.123654097
nbody	$*$	*energy*	[−0.169289380, −0.169289351]	−0.169289351
triple	$*$	AJ	[−40.967014313, −40.966991425]	−40.967002869
adam	$*$	$W0$	[−0.728196859, −0.728196740]	−0.728196859
crout	$[10 \times 10]$	$A[0]$	[1.282013535, 1.282219410]	1.282117963
choleski	$[15 \times 15]$	$A[11][11]$	[4.187705517, 4.187705994]	4.187705994
ldl	$[15 \times 15]$	$A[11][10]$	[0.102230683, 0.102230750]	0.102230720

Table 2: Volatility for the benchmarks

These experiments demonstrate that, for the given test suite, the matrix programs offer a high degree of robustness against platform uncertainties: the value computed for the given variable is guaranteed to fall in the given volatile bound interval (soundness), no matter what platform the code is executed on, and these intervals are quite small.

References

1. IEEE Standards Association. IEEE standard for floating-point arithmetic, 2008. `http://grouper.ieee.org/groups/754/`.
2. Martyn J Corden and David Kreitzer. Consistency of floating-point results using the Intel® compiler, 2010. `http://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf`
3. Yijia Gu and Thomas Wahl. Stabilizing floating-point programs using provenance analysis. In *Verification, Model Checking, and Abstraction Interpretation*, 2017.