Available online at www.sciencedirect.com
ScienceDirect
Mathematics and Computers in Simulation 166 (2019) 451–460
MATHEMATICS AND COMPUTERS IN SIMULATION
www.elsevier.com/locate/matcom

Original articles

Identifying volatile numeric expressions in numeric computing applications☆

Mahsa Bayati[a], Miriam Leeser[a,*], Yijia Gu[b], Thomas Wahl[b]

[a] *Department of ECE, Northeastern University, Boston, MA, United States*
[b] *Khoury College, Northeastern University, Boston, MA, United States*

Received 24 January 2018; received in revised form 14 June 2019; accepted 28 June 2019
Available online 15 July 2019

Abstract

The results of numerical computations with floating point numbers depend on the execution platform, which we define as the hardware and the tools (compilers, etc.) supporting that hardware. One reason for the dependence is that compilers have significant freedom in deciding how to evaluate a floating point expression, as such evaluation is not standardized (not even in standards such as IEEE-754). Another reason is that hardware may or may not provide specialized instructions like Fused Multiply Add (FMA), and if it does, the compiler can take advantage of FMA functionality in different ways.

We call an expression *volatile* if, for some input, its value differs across platform parameters. Differences can become particularly large across heterogeneous parallel architectures. This undermines the software portability promised by programming standards such as OpenCL and significantly impacts reproducibility of results in general.

In this paper, we present a technique that predicts bounds on the output of a program containing volatile expressions when executed on different platforms. Using randomly selected inputs, we compare the bounds to results from running the code across a variety of platforms including CPUs and GPUs. Our results show that the theoretical bounds are relatively tight (within an order of magnitude) and can help users pinpoint where results should be *stabilized*, for instance by restricting expression reordering.

Keywords: Floating point; Heterogeneous computing; Reproducibility; Numerical expressions; OpenCL

1. Introduction

Programming language standardization efforts are often at odds with the goal of permitting efficient language implementations on different computing platforms. Depending on the programming paradigm, a suitable compromise must be found. For floating-point arithmetic, a widely used such compromise is the IEEE 754 floating-point standard. Despite compliance with this standard, the same program run on the same inputs on different platforms can easily produce different results. (We define *platform* as the combination of hardware architecture and the

☆ This work has been partially supported by the US National Science Foundation through Awards CCF-1218075 and CCF-1718235.
* Corresponding author.
E-mail addresses: mbayati@coe.neu.edu (M. Bayati), mel@coe.neu.edu (M. Leeser), guyijia@ccs.neu.edu (Y. Gu), t.wahl@northeastern.edu (T. Wahl).

https://doi.org/10.1016/j.matcom.2019.06.016

compiler that targets it.) Compiler writers have several degrees of freedom when generating code to run on different hardware, including: (i) the compiler may reorder expressions, which affects the values of results because floating point operations are not associative and (ii) a (sub-)expression of the form $a * b + c$ can be translated into a single *fused-multiply add* (FMA) instruction, where the intermediate rounding after the multiplication is avoided. An architecture may support FMA; even if it does, the compiler has freedom regarding which parts of an expression it applies FMA to, and in what order (e.g., in expressions like $a * b + c * d$). These compiler freedoms can lead to volatile expressions across different platforms and such volatility is frequently observed.

Despite this known issue with programs that make use of floating point computations, there are very few tools to help a user of such programs predict the level of volatility and take measures where such volatility is an issue. In this paper we provide a technique that identifies volatile expressions and provides bounds for the values that expressions will exhibit across target platforms. The method is dynamic, i.e. it is based on program execution and relies on concrete inputs provided. We apply it to several applications from the Scalable Heterogeneous Computing (SHOC) Benchmark Suite [10].

In addition to the theoretical approach, we experiment with the same inputs on different hardware platforms. Our results show that the technique consistently produces fairly tight bounds, and that actual results across platforms fall within these bounds. This technique provides useful input to the user of these programs regarding the magnitude of the differences that can be expected, and allows the user to judiciously apply techniques to the portions of the code where such volatility should be limited. This will allow users to improve the reproducibility of their floating point code while minimizing the impact on code performance.

2. Methodology

2.1. Fundamental concepts

Numerical results are affected by the availability of FMA and by decisions the compiler makes about expression evaluation. We refer to the combination of hardware and compiler evaluation decisions as an *expression evaluation model*, denoted M. Throughout this paper we use $+$, $*$ for real addition and multiplication, and $\oplus$, $\otimes$ for floating point addition and multiplication. Thus, the fused multiply–add operation is defined as $fma(a, b, c) = (a * b) \oplus c$: an operation with only one rounding step, in the final addition. For a given input, we say that the result of a numerical program is *reproducible* if we can get bitwise identical results under different evaluation models. For programs computing floating point values, full reproducibility cannot be obtained in general. The largest difference characterizes the extent to which the program is affected by different evaluation models.

Definition 1. Let $r(I, M)$ be the result computed in variable r for program input I under evaluation model M. The **volatility** of r for input I is

$$\mathcal{V}(r, I) = \max_M r(I, M) - \min_M r(I, M) \ . \tag{1}$$

The interval $[\min_M r(I, M), \max_M r(I, M)]$, called **(lower and upper) volatile bounds** of r for input I, contains the results under all possible evaluation models. When $\mathcal{V}(r, I)$ equals 0, the value of r is fully reproducible (platform independent).

Interesting expressions for us are those whose values, for some input, depend on the evaluation model:

Definition 2. Expression Ψ is **volatile** if there exists I s.t. $\mathcal{V}(\Psi, I) > 0$.

We consider evaluation models related to the non-associativity of $\oplus$ and $\otimes$, and the use or non-use of FMA. This set of evaluation models affects the computation of polynomial floating point expressions. Syntactically, a *polynomial volatile expression* is an unparenthesized expression of the form:

$$\Psi \quad :: \quad x_{11} \otimes x_{12} \ldots \otimes x_{1n} \ \oplus \ \ldots \ \oplus \ x_{m1} \otimes x_{m2} \ldots \otimes x_{mn} \tag{2}$$

That is, subexpressions of Ψ are sums of products of floating point variables; each product is called a *monomial*. This form includes many common expressions, such as chains of additions or multiplications, and dot products.

$$\begin{array}{llll} \{v_{11},\ldots,v_{1m}\} & v_1 & := & e_1(I); \\ & & \vdots & \\ \{v_{j1},\ldots,v_{jm}\} & v_j & := & e_j(U_j); \\ & & \vdots & \\ \{r_1,\ldots,r_m\} & r & := & e_r(U_r); \end{array} \quad \rightarrow \quad \begin{array}{llll} [\downarrow v_1, \uparrow v_1] & v_1 & := & \tilde{e}_1(I); \\ & & \vdots & \\ [\downarrow v_j, \uparrow v_j] & v_j & := & \tilde{e}_j(U_j); \\ & & \vdots & \\ [\downarrow v_r, \uparrow v_r] & r & := & \tilde{e}_r(U_r); \end{array}$$

Fig. 1. Concrete semantics (left) and abstract interval semantics (right). The set of possible values under volatility is shown to the left of each statement.

2.2. Background: Numerical abstract interpretation

For a given input I, the execution of numerical program P under evaluation model M can be described by a computational process shown below:

$$\begin{array}{lll} v_1 & := & e_1(I, M); \\ & \vdots & \\ v_j & := & e_j(U_j, M); \\ & \vdots & \\ r & := & e_r(U_r, M); \end{array} \tag{3}$$

where I and r are the input and output of P, respectively; v_j is an intermediate variable; e_j is some floating point expression (in this paper we consider polynomial expressions (2), divisions $\oslash$, or square root *sqrt*); and U_j is the argument vector of e_j (each element of U_j is an input variable, a constant, or an intermediate variable).

A naive way to compute the volatile bounds of r, i.e. the value of the interval $[\min_M r(I, M), \max_M r(I, M)]$, is to calculate the values of r for each of the finitely many different evaluation models. This is, of course, infeasible or very inefficient at best, due to the large number of possible evaluation models. Instead, our technique is based on *abstract interpretation* [9] over the *interval domain* [15], which we review here briefly.

The set of possible values of program variables under all evaluation models is known as the *concrete semantics* of numerical program P from Eq. (3) and shown in the left part of Fig. 1. Here, each v_{ji} represents the value of v_j under evaluation model M_i; each e_j represents the concrete semantics of the jth program statement, also referred to as the *concrete transfer function* of P, in abstract interpretation terminology.

Instead of working with such potentially large sets for each j, we define the *interval abstract semantics* as intervals $[\downarrow v_j, \uparrow v_j]$ as shown in the right part of Fig. 1. Here, $\downarrow v_j$ and $\uparrow v_j$ refer to the minimum and maximum values of v_j; the intervals thus satisfy, for each j, the invariant,

$$\{v_{j1},\ldots,v_{jm}\} \quad \subseteq \quad [\downarrow v_j, \uparrow v_j]\ . \tag{4}$$

The interval is hence a sound (i.e., over-)approximation of $\{v_{j1},\ldots,v_{jm}\}$.

Function $\tilde{e}_j$ in Fig. 1, known as the *abstract transfer function*, implements the abstract semantics of the program statement, i.e. the effect of applying program statements to intervals, rather than concrete values. Functions $\tilde{e}_j$ must be designed such that invariant (4) is maintained. Specifically, the following relationship holds between concrete and abstract transfer functions:

Theorem 3. *Given a floating-point expression $e(I)$ over an input vector $I := (v_1,\ldots,v_n)$, let $\mathbb{I} = [\downarrow v_1, \uparrow v_n] \times \cdots \times [\downarrow v_n, \uparrow v_n]$, and let $[\downarrow v, \uparrow v]$ be the abstract output interval, i.e. the result of applying $\tilde{e}$ to $\mathbb{I}$. Then:*

$$\left[\min_{I\in\mathbb{I}} \min_M e(I, M), \max_{I\in\mathbb{I}} \max_M e(I, M)\right] \quad \subseteq \quad [\downarrow v, \uparrow v]\ .$$

In the next section, we show how to design abstract transfer functions $\tilde{e}$ for polynomial volatile expressions that satisfy this requirement. Note that transfer functions for non-volatile expressions do not need to take volatility into account and can thus be derived from standard interval abstraction for floating point arithmetic [15].

2.3. Abstract transfer functions for volatility

Given abstract input $\mathbb{I}$, our technique to approximating the lower volatile bound for an expression Ψ, as in (2), consists of two steps (the upper bound is calculated similarly). First Algorithm 2.1 transforms each monomial $x_{i1} \otimes x_{i2} \ldots \otimes x_{in}$ into two-constant form $c_i^{\dashv} \otimes c_i^{\vdash}$. The form returned consists of an upper and a lower term, which when multiplied produce an answer within the range of values of the solution. The reason for choosing this form is that it allows us to consider alternative ways of applying FMA between adjacent monomials.

Algorithm 2.1: Find the two-constant form of monomial m

Input: $m := v_1 \otimes \ldots \otimes v_n$, where v_i can be a constant or a variable
$\downarrow m = +\infty$;
for $(c_1, \ldots, c_n) \in \{\downarrow v_1, \uparrow v_1\} \times \ldots \times \{\downarrow v_n, \uparrow v_n\}$ **do**
  $(t^{\dashv}, t^{\vdash}) = getMin_{mul}(c_1, \ldots, c_n)$;
  if $\downarrow m > t^{\dashv} * t^{\vdash}$ **then**
    $\downarrow m = t^{\dashv} * t^{\vdash}$;
    $c^{\dashv} = t^{\dashv}$;
    $c^{\vdash} = t^{\vdash}$;
  end
end
return $(c^{\dashv}, c^{\vdash})$;

Function $getMin_{mul}$ in Algorithm 2.1 calculates a pair of floating point numbers whose product is the minimum value of the monomial. The function is defined as

$$getMin_{mul}(c_1, \ldots, c_n) = (A[1, L[1, n]], A[L[1, n] + 1, n])$$

where $A[i, i] = c_i$, $A[i, j] = A[i, L[i, j]] \otimes A[L[i, j] + 1, j]$ for $i < j$, and

$$L[i, j] = \begin{cases} \underset{k:i \le k < j}{\text{argmin}} \mid A[i, k] * A[k + 1, j] \mid & \text{if } sign(m) = + \\ \underset{k:i \le k < j}{\text{argmax}} \mid A[i, k] * A[k + 1, j] \mid & \text{if } sign(m) = - \ . \end{cases}$$

Function $sign(m)$ returns the sign of the multiplication result of the monomial. Note that we use real multiplication in the definition of $L[i, j]$ instead of $\otimes$ as in the definition for $A[i, j]$: the multiplication in FMA is conceptually done in real arithmetic.

A separate pair of floating point numbers whose product is the maximum value of the monomial is computed using a similar algorithm, replacing $getMin_{mul}$ with an analogous function $getMax_{mul}$. After obtaining the two-constant form, the polynomial expression has the shape of a standard dot product: $c_1^{\dashv} \otimes c_1^{\vdash} \oplus \ldots \oplus c_n^{\dashv} \otimes c_n^{\vdash}$.

In the second step we obtain the lower volatility bound $\downarrow \Psi$ of the dot product expression $\Psi = v_{11} \otimes v_{21} \oplus \ldots \oplus v_{1n} \otimes v_{2n}$ (this form is guaranteed by step 1), under evaluation models that support FMA, as follows. We need to consider not only different ways of applying FMA, but at the same time different ways of parenthesizing the expression. For example, for $n = 3$ the above expression can be evaluated in many different ways, including

$$v_{11} \otimes v_{21} \oplus (v_{12} \otimes v_{22} \oplus v_{13} \otimes v_{23})$$
$$fma(v_{11}, v_{21}, v_{12} \otimes v_{22} \oplus v_{13} \otimes v_{23})$$
$$fma(v_{11}, v_{21}, fma(v_{13}, v_{23}, v_{12} \otimes v_{22}))$$
$$fma(v_{13}, v_{23}, fma(v_{12}, v_{22}, v_{11} \otimes v_{21}))$$

Our method for computing $\downarrow \Psi$ is to determine the value $B[1, n]$, for the array B defined as

$$B[i, j] = \begin{cases} v_{1i} \otimes v_{2i} & \text{if } i = j \\ \min \{fma(v_{1i}, v_{2i}, B[i + 1, j]), & \\ \quad \min\limits_{i<k<j-1} \{B[i, k] \oplus B[k + 1, j]\}, & \\ \quad fma(v_{1j}, v_{2j}, B[i, j - 1])\} & \text{if } i < j \end{cases}$$

In a similar way, we obtain the maximum value, $\uparrow \Psi$. We summarize the guarantee of the volatile bounds as follows:

Theorem 4. *Given interval input* $\mathbb{I}$*, let* $[\downarrow \Psi(\mathbb{I}), \uparrow \Psi(\mathbb{I})]$ *be the interval resulting from our analysis. Then:*

$$[\min_{I\in\mathbb{I}} \min_{M} \Psi(I, M), \max_{I\in\mathbb{I}} \max_{M} \Psi(I, M)] \subseteq [\downarrow \Psi(\mathbb{I}), \uparrow \Psi(\mathbb{I})]$$

Applying Theorem 4 inductively across all steps of the program that lead to the final result r, we obtain, for any input I:

$$[\min_{M} r(I, M), \max_{M} r(I, M)] \subseteq [\downarrow r(I), \uparrow r(I)] \ .$$

2.4. Implementation

We have implemented the above techniques in a runtime library, the core of which is a customized datatype `ifloat`, that performs floating point operations *abstractly*. That is, it takes floating point intervals as input (where concrete values are taken as singleton intervals) and produces floating point interval outputs by implementing the above abstract transfer functions $\tilde{e}$. For each program variable, `ifloat` thus tracks its volatile bounds. The abstract program execution is slower than the concrete (single-value) execution (see Section 4). To apply our analysis to a given program, the user needs to make the following two changes to the program: they need to replace all native `float` types with `ifloat`, and they need to assign parenthesized parts of polynomial expressions (which prevent compilers from reordering) to a temporary variable to force this evaluation order. The automation of these steps is left for future work. In our experiments we use single-precision float as the numeric data type and, as in most programs, *round-to-nearest-ties-to even* as the rounding mode.

3. Experiments and results

3.1. Experimental setup

In our experiments we investigate the volatile bounds for a number of application programs. We show results for Ray Tracing, Stencil Computations (SC) and Molecular Dynamics (MD). Ray tracing was chosen as a small example that illustrates the issues we are addressing. Stencil and MD are taken from the SHOC Benchmark Suite [10]. Molecular Dynamics is used for applications such as drug discovery where users are concerned about getting consistent results. Stencil Computations are the basis of many large applications, so here, too, consistent results are important. For SC and MD, we choose inputs at random, but keep these inputs constant across all experiments. We run the volatility analysis on these inputs and then run experiments across a number of platforms to illustrate the volatility and to show that the theoretical analysis does indeed produce tight bounds. We also analyze how differences grow with the input sizes and the number of iterations. Input data sets and instructions for running the code are available at our project website [3] as well as supplementary material for this paper. This provides sufficient information for the interested reader to reproduce our results. In the future we will consider allowing the user of our tool to suggest inputs or intervals to ensure meaningful values are covered, or to steer the analysis towards inputs that are known to give rise to large volatility.

Platforms. For our experiments we target a range of different computer hardware (all compliant with IEEE 754–2008). We run our experiments on both CPUs and GPUs, from the major manufacturers (AMD, Intel, and NVIDIA) representing a range of technology families and complexity (see Table 1). Applications are implemented in OpenCL and each targeted hardware platform has an OpenCL compiler provided by its manufacturer. We use OpenCL as it allows us to run the same code on different hardware. Our techniques are generally applicable to other languages, especially those based on C.

3.2. Ray tracing

As a small example we use Raytracing code taken from http://www.cc.gatech.edu/~phlosoft/photon/. The relevant snippet of this code is shown in Fig. 2. For these experiments we choose inputs that trigger a control flow instability;

Table 1
Target computer hardware.

	Type	Manuf.	Description	Year	FMA
1	CPU	Intel	E5−2650	2016	Y
2	CPU	AMD	A10−6700	2013	N
3	GPU	NVIDIA	Tesla K20	2012	Y
4	GPU	NVIDIA	Tesla K40	2013	Y
5	GPU	NVIDIA	Tesla K80	2014	Y
6	GPU	AMD	Radeon HD 7660G	2013	N

Table 2
Raytracing: different D values obtained experimentally; analytic volatility bounds.

	AMD CPU	AMD GPU	Intel	NVIDIA	Analytic D bounds
set1	+1.441E−1	+1.250E−1	+1.441E−1	−1.058E−1	[−2.528E−1, 1.567E−1]
set2	+1.086E−2	+0.000E+0	+1.086E−2	−3.426E−2	[−6.888E−2, 6.389E−2]
set3	−4.545E+0	−8.000E+0	−4.545E+0	+6.126E+0	[−1.600E+1, 9.205E+0]
set4	+3.874E+0	+0.000E+0	+3.874E+0	−4.715E+0	[−8.000E+0, 1.685E+1]
set5	+4.802E+0	+0.000E+0	+4.802E+0	−9.0989E+0	[−1.6023E+1, 5.021E+1]
set6	+9.767E−2	+0.000E+0	+9.767E−2	−3.902E+0	[−4.852E+0, 2.305E+0]
set7	+1.953E−1	+2.500E−1	+1.953E−1	−3.041E−3	[−5.031E−1, 3.736E−1]

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

Fig. 2. Code for ray tracing.

in this code this occurs for `if (D > 0)`. The results can cause the code after this branch to be executed or not depending on platform, thus exhibiting a control flow instability [13]. We are interested in the value of D, given inputs s, r, and *radiusSq*. Table 2 shows the value of D computed for seven input sets. In all cases, the observed results lie within the computed volatility bounds. In all input sets 1 through 7, a control flow instability is observed.

3.3. 2DStencil

2DStencil is implemented as part of SHOC [10]. We use a 9-point stencil pattern. We generate the input matrix data randomly and run the algorithm on different platforms using the same data. The output is a matrix with the same size as the input matrix. We report the element which exhibits the maximum absolute volatility in our results. We compare the observed differences to those found using the technique described in Section 2.1. These were applied to two input sets of size 128 × 128 with 50 iterations and one input set of size 512 × 512 with 10 iterations. Our results show that the analysis does indeed provide accurate bounds which are approximately ten times larger than the largest observed difference. Fairly tight bounds help users identify where the differences arise in floating point computations. We also observe that NVIDIA GPUs and Intel CPU produce no difference in results, and AMD GPUs and AMD CPUs also show no difference. The largest difference for Stencil2D on all platforms is between an NVIDIA GPU and an AMD GPU. These differences are shown in Table 3.

We also study how the differences scale in multiple dimensions, including size of the input data and number of iterations. Tables 4 and 5 show these differences across the NVIDIA and AMD GPU platforms. It can be concluded

Table 3
Stencil: Largest absolute difference and volatility bounds.

Input size — # of iterations	Max. Abs. differences observed	Analytic diff.
set1: 128 × 128 — 100	7.50E−1	1.087E+1
set2: 128 × 128 — 50	2.38E−6	2.336E−5
set3: 512 × 512 — 10	9.37E−2	3.437E−1

Table 4
Stencil2D results: Differences AMDGPU/NVIDIA for 128 × 128 matrix.

128 × 128	10 iter	100 iter	250 iter	500 iter	750 iter	1000 iter
Max Abs. Diff	3.9E−3	7.50E−1	1.92E+3	5.368E+8	1.363E+14	3.17E+19
Relative Diff	2.7E−7	4.97E−7	1.01E−6	1.656E−6	2.440E−6	3.308E−6

Table 5
Stencil2D results: Differences AMDGPU/NVIDIA for 1024 × 1024 matrix.

1024 × 1024	10 iter	100 iter	250 iter	500 iter	750 iter	1000 iter
Max Abs. Diff	3.75E−1	8.000E+1	1.802E+5	6.87E+10	1.857E+16	4.722E+21
Relative Diff	3.30E−7	6.089E−7	1.005E−6	1.85E−6	2.64E−6	3.339E−6

Table 6
MD: Largest absolute difference and analytic volatility for 12 288 atoms.

Platforms	Max. Abs. differences observed	Analytic volatility
NVIDIA vs. AMDGPU	4.503E16	
NVIDIA vs. AMDCPU	3.298E12	
NVIDIA vs. Intel	3.298E12	3.6705E17
Intel vs. AMDGPU	4.503E16	
Intel vs. AMDCPU	1.099E12	
AMDGPU vs. AMDCPU	4.503E16	

Table 7
Scaling of MD results differences AMDGPU vs. NVIDIA.

No. of atoms	24 576	36 864	73 728
Max Abs. Diff	4.503E+16	1.441E+17	1.441E+17
Relative Diff	3.892E−07	7.790E−07	7.790E−07

that the absolute differences grow as the number of iterations and the input size grow, but the relative error remains at 10^{-7}.

3.4. Molecular dynamics

Molecular Dynamics (MD) [10] is the computation of the Lennard-Jones potential. The program input is the Cartesian position (x, y, z) of a number of atoms and the forces are calculated in each direction (x, y, z) for each atom based on accumulation of the forces from neighboring atoms within a specific cut-off distance. The output is the force applied to each atom, in (x, y, z) coordinates. In our experiments, we run the MD application for a specific number of atoms with their positions randomly generated, use 128 neighboring atoms and a cut-off distance of 16.0. Table 6 shows volatility bounds based on dynamic analysis and the differences observed between different runs for 12288 atoms. Table 7 shows that the maximum absolute difference increases as the number of atoms increases, however the relative difference remains constant. Each platform generates results which differ from all others, but the largest differences are observed between the AMD and NVIDIA GPUs, so these are the ones we report.

4. Discussion

As our experimental results demonstrate, platform dependence of numeric computations is a bigger problem for some applications than for others. Where does one draw the line — what constitutes an acceptable platform dependence? This is a non-trivial research problem by itself; the answer depends on the way the numeric results are used.

A software pattern that is generally cause for concern in the context of reproducibility is control flow that depends on numeric results. If there are inputs for which the control flow decision is close to the decision boundary, then the flow of control may depend on the computational platform — for identical inputs, the program may execute different segments of code on different machines. Such changes in the behavior of programs across platforms are usually a serious indicator. They have ramifications for the more general class of *decision-making* programs, such as image classifiers, which are beyond the scope of this paper. We have demonstrated, using the ray tracing example, that this is a realistic possibility: in this program the volatility in variable D makes execution of the conditional code platform-dependent.

Such uncertainties clearly must be addressed. In separate work, we have proposed ways to reduce platform uncertainty [12]. A naive way to achieve this 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 [8]. In [12] we present a more fine-grained approach that, given some target expression E, aims to stabilize only *some* evaluation aspects of *some* preceding statements, namely of those that contribute most to the uncertainty in E. Our technique returns information on what these statements are, and what kinds of uncertainties in their evaluation (ordering, FMA, etc.) are to blame for E's non-reproducibility. This allows the user to apply fine-grained, local code stabilization, with minimal impact on program performance.

4.1. Related work

Much prior work in analyzing numerical programs focuses on rounding errors. Interval analysis [15] calculates an interval for each variable during execution that guarantees to contain the rounding error bound of the variable. Although we do not target rounding errors in this paper, we use interval analysis as well to soundly approximate the volatile bounds of each variable. In general, a drawback of interval analysis is that the resulting intervals are often rather pessimistic (large), owing to the fact that the analysis ignores relations among variables in the expressions. This issue can be addressed in two ways: either by interval splitting, so that the initial intervals are very small, thereby reducing the abstract arithmetic error to propagation error, or by using more precise *relational* domains, such as the octagon domain. In our work, however, we have found in the experiments that the coarseness of interval approximation is not a bottleneck. The reason is that we only run the analysis on single inputs (so the abstract inputs are singleton intervals, rather than true input ranges), which we can think of as maximally split intervals.

Besides the above direct analysis of rounding errors of the program, research is also being conducted on detecting deviations of numeric programs' behaviors compared to idealistic exact-arithmetic programs, caused by rounding errors. Existing works include detecting floating point exceptions [2], and instable control-flow paths [1,7].

The work most closely related to reproducibility analysis in this paper is research about IEEE compliance problems [14] encountered in actual heterogeneous computing environments (CPU/GPU/FPGA). Boldo et al. [4] present a formally verified `C` compiler that guarantees IEEE-compliant floating point machine code, which is achieved by enforcing "a single way to compile", akin to strict semantics. This ensures code stability, but does not address the question of whether there is any significant instability in the program in the first place, or how much instability is tolerable in the program. Other works [5,6,16] prove a *maximum rounding error* under platform variations. The approach is deductive (proof-based) and not suitable for identifying inputs that cause platform dependence. Our dynamic analysis technique is able to identify inputs (among the given test set) with critical stability problems, and generally provides an upper bound on how poorly the program may behave, in terms of reproducibility, when run on platforms other than the development machine.

4.2. Planned improvements

The code linked against our `ifloat` library (Section 2.4) is currently up to three orders of magnitude slower than the original code with single precision. This is due to the extra information tracked by our library, and also

the extensive use of rational numbers in the process; the analysis itself is done in exact arithmetic, using the GNU Multiple Precision arithmetic library (https://gmplib.org). For large numerical programs, the use of rational arithmetic quickly becomes the performance bottleneck. Similar to the approach applied by Fluctuate [11], we plan to offer the choice between an analysis performed in floating point or rational arithmetic.

The overapproximation necessary to compensate for the rounding error in the analysis itself (not in the program) will affect the precision of the analysis. To alleviate this issue, we require that the precision used in the floating point based analysis is higher than that of the analyzed code. If necessary, this can be achieved using variable-precision floating point libraries such as GNU MPFR (http://www.mpfr.org).

5. Conclusions and future work

We presented a technique for identifying the bounds of floating point expressions due to differences that numeric programs exhibit, for the same input, across computational platforms. These differences arise due to lax expression evaluation rules in most programming languages (whether complying with the IEEE 754 floating point standard or not), as well as differences in floating-point hardware. We also presented the results of experiments that show that the volatile bounds produced are tight and that these differences occur not only for carefully selected critical inputs, but also for randomly chosen ones. We show how these results scale with iterations and input sizes and provide techniques for identifying when such differences may be considered critical by the user.

Volatility in programs arises not only as a result of platform changes, when programs are ported from one architecture to another: the same phenomenon is responsible for making many floating-point *compiler optimizations* unsound. Compilers frequently fold constants or reorder expressions, which often causes the program results to change on some inputs. This phenomenon is known; programmers are warned by compilers (at least hidden somewhere in the manual page) that *fastmath* optimizations do not guarantee input/output equivalent expressions. Little is known about how to make optimizations value-stable, especially while preserving as much of the optimization benefits as possible, such as improved code performance. Since the underlying root cause is the same as in platform-dependent computations, namely non-invariance of floating-point expression values under reorderings and other "apparently equivalent" code transformations, there is hope to assist compilers in optimizing programs more predictably using some of the techniques reported on in this paper.

Acknowledgments

This material is based upon work supported by the US National Science Foundation. We would like to thank the reviewers for helping to improve the paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.matcom.2019.06.016.

References

[1] T. Bao, X. Zhang, On-the-fly detection of instability problems in floating-point program execution, in: ACM SIGPLAN International Conference on OOPSLA 2013, Part of SPLASH 2013, Indianapolis, USA, 2013, pp. 817–832.

[2] E.T. Barr, T. Vo, V. Le, Z. Su, Automatic detection of floating-point exceptions, in: 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Rome, Italy, 2013, pp. 549–560.

[3] M. Bayati, M. Leeser, Y. Gu, T. Wahl, Identifying volatile numeric expressions in OpenCL applications, http://www.coe.neu.edu/Research/rcl/projects/FPReproducibility/matcom.html, 2018.

[4] S. Boldo, J. Jourdan, X. Leroy, G. Melquiond, A formally-verified C compiler supporting floating-point arithmetic, in: ARITH 2013, Austin,TX, USA, 2013, pp. 107–115.

[5] S. Boldo, T.M.T. Nguyen, Hardware-independent proofs of numerical programs, in: Second NASA Formal Methods Symposium (NFM) 2010, Washington D.C., USA, 2010, pp. 14–23.

[6] S. Boldo, T.M.T. Nguyen, Proofs of numerical programs when the compiler optimizes, ISSE 7 (2) (2011) 151–160.

[7] W. Chiang, G. Gopalakrishnan, Z. Rakamaric, Practical floating-point divergence detection, in: Languages and Compilers for Parallel Computing, 2015, pp. 271–286.

[8] M.J. Corden, D. Kreitzer, 2010. Consistency of Floating-Point Results using the Intel® Compiler or Why doesn't my application always give the same answer?, http://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf, last accessed March 2019.

[9] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: ACM Symposium on Principles of Programming Languages, California, USA, 1977, pp. 238–252.
[10] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, J. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite. https://github.com/vetter/shoc/wiki, last accessed March 2019.
[11] E. Goubault, Static analyses of the precision of floating-point operations, in: Static Analysis, 8th International Symposium, SAS 2001, Paris, France, 2001, pp. 234–259.
[12] Y. Gu, T. Wahl, Stabilizing floating-point programs using provenance analysis, in: VMCAI, 2017, pp. 228–245.
[13] Y. Gu, T. Wahl, M. Bayati, M. Leeser, Behavioral non-portability in scientific numeric computing, in: EURO-PAR, 2015, pp. 558–569.
[14] D. Monniaux, The pitfalls of verifying floating-point computations, ACM Trans. Program. Lang. Syst. 30 (3) (2008) 12:1–12:41.
[15] R.E. Moore, R.B. Kearfott, M.J. Cloud, Introduction to Interval Analysis, SIAM, 2009.
[16] T.M.T. Nguyen, C. Marché, Hardware-dependent proofs of numerical programs, in: Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, 2011, pp. 314–329.