Stabilizing Floating-Point Programs Using Provenance Analysis

Yijia Gu(✉) and Thomas Wahl

College of Computer and Information Science, Boston, USA
{guyijia,wahl}@ccs.neu.edu

Abstract. Floating-point arithmetic is a loosely standardized approximation of real arithmetic available on many computers today. Architectural and compiler differences can lead to diverse calculations across platforms, for the same input. If left untreated, platform dependence, called *volatility* in this paper, seriously interferes with result reproducibility and, ultimately, program portability. We present an approach to *stabilizing* floating-point programs against volatility. Our approach, dubbed *provenance analysis*, traces volatility observed in a given intermediate expression E back to volatility in preceding statements, and quantifies individual contributions to the volatility in E. Statements contributing the most are then stabilized, by disambiguating the arithmetic using expression rewriting and control pragmas. The benefit of *local* (as opposed to program-wide) stabilization is that compilers are free to engage performance- or precision-enhancing optimizations across program fragments that do not destabilize E. We have implemented our technique in a dynamic analysis tool that reports both volatility and provenance information. We demonstrate that local program stabilization often suffices to reduce platform dependence to an acceptable level.

1 Introduction

Floating-point arithmetic (FPA) is a loosely standardized approximation of real arithmetic available on many computers today. The use of approximation incurs commonly underestimated risks for the reliability of embedded software. One root cause for these risks is the relatively large degree of freedom maintained in the most widely adopted FPA standardization, IEEE 754 [1]: 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 calculations more or less at will.

The price we pay for these freedoms is reduced reproducibility of results, especially for software that is run on diverse, possibly heterogeneous hardware. For example, distributing a computation across nodes in a cluster may rearrange the code in ways that produce results very different from what was observed in the comfort of the office PC environment. This platform dependence of results,

Work supported by US National Science Foundation grant no. CCF-1253331.

A. Bouajjani and D. Monniaux (Eds.): VMCAI 2017, LNCS 10145, pp. 228–245, 2017.
DOI: 10.1007/978-3-319-52234-0_13

called *volatility* in this paper, translates de facto into non-portability and, ultimately, inferior reliability. Examples of *discrete* program behaviors that may be affected, for certain inputs, when moving from one platform to another include the program's control flow, and invariants that hold on some platforms but not others [8,14,16]. Problems of this nature are hard to detect using traditional testing, as most developers cannot afford to run their programs on a multitude of platforms and compare the results.

In this paper we present an approach to identifying such problems *without* the need to run the given program on diverse platforms. Our approach employs a dynamic analysis that executes the program on inputs of interest, such as from a given test suite. For each input I and each intermediate program expression E, we compute a (tight) interval, called the *volatile bound*, such that the value of E on input I is guaranteed to be contained in that interval, *no matter how the program is compiled* on the path to E, subject only to IEEE 754 compliance of the platform. The volatile bound interval is computed via an abstract domain that takes any possible expression evaluation order into account, as well as the possibility of FMA contraction, for every expression on the path to E (not only for E itself). Our analysis technique issues a warning when the observed volatility becomes critical, for example when E is of the form `c < 0` and 0 is inside the volatile bound for c: this means the comparison is *unstable* for the given input—the subsequent control flow depends on the execution platform.

Our technique then goes a significant step further and proposes ways to fix this instability. A naive way is to "determinize" the compilation of the entire program, using compiler flags that enforce "strict evaluation", such as `/fp:strict` for Visual Studio `C++`. This unfortunately destroys optimizations that compilers can apply to harmless (stable) fragments of the code; it may thus needlessly reduce a program's precision and efficiency [5]. We propose a more fine-grained approach that aims to stabilize only *some* evaluation aspects, of *some* statements S that contribute *most* to the instability in the target expression E. We call the information of what these statements are the *provenance* of E's instability. Provenance information also includes what kinds of ambiguities in S's evaluation are responsible for E's instability, i.e. evaluation order or the potential for FMA application. This allows very fine-grained, local code stabilization, after which the user can repeat the analysis, to determine whether the critical instability in E has disappeared.

We have implemented our technique in a library called `ifloat`. Given a program and a test suite, the goal of the library is to stabilize the program against expression volatility, for all inputs, using provenance analysis. It is immaterial on what platform the analysis itself is executed. We conclude the paper with experiments that illustrate on a number of benchmarks how the volatility of critical expressions diminishes as local stabilization measures are applied. We demonstrate the high precision of our (necessarily approximate) analysis compared to an idealistic but unrealistic one that compiles and runs the program twice—with and without stabilization measures.

2 A Motivating Example

We use the `C` program fragment shown in Listing 1.1 as an example to illustrate our approach. The code is used in *Ray Tracing* applications[1] in computational geometry. Consider the input

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

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)
    ...; }
```

Listing 1.1. Ray Tracing

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

int raySphere(ifloat *r, ifloat *s,
     ifloat radiusSq) {
  ifloat A, B, C, D;
  A = dot3(r,r);                 // l1
  B = -2.0 * dot3(s,r);          // l2
  C = dot3(s,s) - radiusSq;      // l3
  D = B*B - 4*A*C;               // l4
  if (D > 0)
    ...; }
```

Listing 1.2. Ray Tracing with `ifloat`

The numeric instability eventually leading to the decision divergence is due to the presence (GPU) or absence (CPU) of FMA hardware instructions on the two processors. FMA is a contraction of floating-point multiplication and addition in expressions of the form `a * b + c`, so that the multiplication is in effect performed without intermediate rounding. Such expressions come up in Listing 1.1 in function `dot3` and in the expression defining `D`.

To analyze and debug the Ray Tracing program for instability issues using our library, we first change all `float` types in the program to `ifloat`. To enable our tool to identify the root cause of the divergence, we add comment labels to each statement in the `raySphere` function that we wish to include in the analysis. The program after these transformations is shown in Listing 1.2.

Compiling and running the transformed program outputs a *volatile bound* of $[-0.252806664, 0.156753913]$ for `D`. This interval overapproximates the set of

[1] http://www.cc.gatech.edu/~phlosoft/photon/.

values any IEEE 754-compliant compiler/hardware combination can possibly produce, for input (1). Due to the size of this interval, and the looming `D > 0` branch, our `ifloat` library generates a warning to tell users that the code may have instability (platform dependency) problems.

To fix the instability problem, one could now simply "determinize" the floating-point compilation of the program. This can be achieved using strict-evaluation compiler flags, such as `/fp:strict` in Visual Studio `C++`, which typically disable FMA and force a specific evaluation order for chained operations. However, there is of course a trade-off between reproducibility on the one hand, and platform-specific precision and performance on the other: program-wide code determinization prevents optimizations that compilers could otherwise apply to harmless (stable) fragments of the code; they may thus needlessly reduce a program's precision and efficiency [5]. Instead, we propose a more fine-grained approach that only fixes *some* evaluation aspects of *select* statements, namely those that affect the comparison, or any other user-provided critical expression. At the end of this section we show, using the Ray Tracing program, how to achieve statement-level fixation in `C++`.

But how do we determine which statements to stabilize? Identifying those merely based on the volatile bounds of the expressions computed in them is insufficient. To see this, we list in Table 1 the volatile bounds of the intermediate variables on the path to the computation of `D`. The size of `D`'s volatile bound clearly dominates that for the other variables, suggesting that we should fix the evaluation of the expression for `D` itself. However, turning off FMA and forcing left-to-right evaluation for the assignment to `D` results in a volatile bound of $[-0.250000000, 0.125000000]$, nearly unchanged from the bound before stabilization. The new bound clearly still permits diverging control flows.

Table 1. Volatile bounds of intermediate variables

Variable	Volatile bound	Strict value
`A`	(601.957031250, 601.957031250)	601.957031250
`B`	(−1129.186889648, −1129.186767578)	−1129.186889648
`C`	(529.548950195, 529.549011230)	529.548950195
`D`	(−0.252806664, 0.156753913)	0.125000000

Instead, our analysis of the transformed program produces information on the *provenance* of the instability of `D`'s value, i.e., for each preceding statement, a measure of how much its instability contributes to that of `D`. In addition, we output the reason of the instability in each statement, to guide the programmer as to what aspect of the evaluation to fix. This output is shown in Table 2.

Consider a pair of the form (L, U) in the "Provenance for `D`" column in the row for label `l1`. The left component L ("lower") specifies by how much the left boundary of the volatile bound interval for `D` shifts, due to the numeric instability 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.

Table 2. Provenance information for `D` and reason for instability contribution

Stmt. label	Provenance 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

The instability 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 of how to do this at the end of this section), our analysis results in a new volatile bound for `D` of $[-0.002806663, 0.156753913]$; the corresponding provenance information is shown in Table 3.

Table 3. Provenance information for the calculation of `D` after partial stabilization

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

From this bound we conclude that the comparison `D > 0` still suffers from instability (which is now, however, less likely to materialize in practice). 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 stabilized version of the Ray Tracing program is shown in Listing 1.3 (differences in red).

```
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)
    ...; }
```

Listing 1.3. Stable version of Ray Tracing, obtained using compiler pragmas and parentheses (`dot3` partially inlined)

In practice, little is gained by stabilizing a numeric computation for a single input. Instead, in our experiments we determine provenance information embedded into a dynamic analysis tool on a test suite that comes with the given program. Given a critical target expression `E`, our tool determines which preceding statement causes the greatest expansion of `E`'s volatile bound, maximized across all inputs. The identified cause of volatility for that statement is then lifted. The analysis repeats until an acceptable level of stability is achieved. In the technical part of this paper (Sects. 4, 5, 6 and 7) we describe details of our provenance analysis for a given input. The application of this analysis across a test suite is discussed in Sect. 8.

3 Background: Volatility in Floating-Point Arithmetic

We use standard symbols like $+$ and $*$ for real-arithmetic operators, and circled symbols like $\oplus$ and $\otimes$ for floating-point operators. The latter are defined by the IEEE 754 standard [1, "the Standard" in this paper], for instance floating-point addition as $x \oplus y = rd(x + y)$, where rd is the rounding function (often refered to as *rounding mode*). The Standard postulates five such functions, all of which satisfy the following **monotonicity property**:

$$\forall x, y \in \mathbb{R} : \; x \leq y \implies rd(x) \leq rd(y) \; . \tag{2}$$

Unlike binary operations, floating-point *expressions*, which feature chains of operations as in $x \oplus y \oplus z$, do not come with a guarantee of reproducibility: compilers have the freedom to evaluate such expressions in any order. It is well known that $(x \oplus y) \oplus z$ and $x \oplus (y \oplus z)$ can return different results, e.g. due to an effect known as *absorption* when x is much larger than y and z. As a result, floating-point addition lacks associativity, as does multiplication.

Other sources of non-reproducibility are differences in the available floating-point hardware. The most prominent example is the *fused multiply-add* (FMA) operation, defined by $fma(a, b, c) = (a * b) \oplus c$. That is, the two operations are performed as if the intermediate multiplication result was not rounded at all. Not all architectures provide this operation; if they do, there is no mandate for the compiler to compute $a \otimes b \oplus c$ via FMA. Worse, expressions like $a \otimes b \oplus c \otimes d$ permit multiple ways of applying FMA.

Definitions and notation. Expressions that suffer from ambiguities due to reordering of $\oplus$ and $\otimes$ expressions and due to (non-)use of FMA are called *volatile* in this paper [14]. Formally, these are parenthesis-free expressions of the form

$$x_{11} \otimes x_{12} \ldots \otimes x_{1n} \oplus \ldots \oplus x_{m1} \otimes x_{m2} \ldots \otimes x_{mn} \; ,$$

which includes chains of addition, chains of multiplication, and dot products.

"Platform parameters" refers to compiler and hardware parameters, namely the availability of FMA, and collectively the decisions made by the compiler about expression evaluation. We refer to an instantiation of such parameters

as an *expression evaluation model* M. We denote by $\Psi(I, M)$ the value of (volatile) expression Ψ on input I and expression evaluation model M. A volatile expression Ψ is *stable on input* I if, for all evaluation models M_1, M_2, we have $\Psi(I, M_1) = \Psi(I, M_2)$.

Relevant for the goal of stabilization in this paper is the *strict* expression evaluation model M_{str}, defined as the model that disables FMA support and evaluates sums and products from left to right. Enforcing this model is supported on many compiler platforms, such as using the `/fp:strict` flag in Visual Studio `C++`. Finally, as this paper is about numeric reproducibility, we treat the occurrence of special floating-point data like NaN or $\pm\infty$ as a failure and abort.

4 Provenance of Volatility: Overview

A volatile expression Ψ can be evaluated under a number of different evaluation models M. In general, this in turn can give rise to as many different results, for a fixed input I. One way to capture these results $\Psi(I, M)$ is using an interval:

Definition 1. *Given volatile expression Ψ and input I, the* ***volatile bound*** $[\downarrow\Psi(I), \uparrow\Psi(I)]$ *is the interval defined by*

$$\downarrow\Psi(I) = \min_M \Psi(I, M) \ , \quad \uparrow\Psi(I) = \max_M \Psi(I, M) \ . \tag{3}$$

The size of the volatile bound characterizes the volatility of Ψ for input I: the larger the bound, the more volatile Ψ. We extend the above definition to input intervals $\mathbb{I}$ over floating-point numbers: $[\downarrow\Psi(\mathbb{I}), \uparrow\Psi(\mathbb{I})]$ is the interval defined by

$$\downarrow\Psi(\mathbb{I}) = \min_{I\in\mathbb{I}} \min_M \Psi(I, M) \ , \quad \uparrow\Psi(\mathbb{I}) = \max_{I\in\mathbb{I}} \max_M \Psi(I, M) \ . \tag{4}$$

However, with this definition the size of the bound no longer characterizes the volatility of the expression. For example, given assignment statement $r = x \oplus y$, if the input intervals for x and y are large, the resulting bound for r will also be large, but only due to the uncertainty in inputs, not different evaluation models.

To be able to distinguish input uncertainty from evaluation uncertainty, we introduce the concept of *volatile error*, which measures the difference between an arbitrary evaluation model and the *strict* evaluation model M_{str} (Sect. 3):

Definition 2. *The* ***volatile error*** *of Ψ on input I is the pair $(\underline{e}, \bar{e})$ with*

$$\underline{e} = \min_M \Psi(I, M) - \Psi(I, M_{str}) \ , \quad \bar{e} = \max_M \Psi(I, M) - \Psi(I, M_{str}) \ . \tag{5}$$

Values $\underline{e}$ and $\bar{e}$ represent the "drift" of Ψ on input I, relative to $\Psi(I, M_{str})$, to the left and right due to different evaluations: we have $\underline{e} \le 0 \le \bar{e}$; an expression's value is platform-independent iff $\underline{e} = \bar{e} = 0$.

Definition 2 lends itself to being extended to the case of input intervals $\mathbb{I}$: in this case, the volatile error of Ψ is the pair $(\underline{e}, \bar{e})$ with

$$\begin{aligned} \underline{e} &= \min_{I\in\mathbb{I}} \min_M \Psi(I, M) - \min_{I\in\mathbb{I}} \Psi(I, M_{str}) \ , \\ \bar{e} &= \max_{I\in\mathbb{I}} \max_M \Psi(I, M) - \max_{I\in\mathbb{I}} \Psi(I, M_{str}) \end{aligned} \tag{6}$$

The pair $(\underline{e}, \bar{e})$ represents the enlargement of the volatile bound purely due to evaluation uncertainties. For instance, for $\Psi := x \oplus y$, we indeed have $\underline{e} = \bar{e} = 0$.

As we have seen in Sect. 2, knowing the volatile error for a single expression is not enough. We are also interested in the provenance (origin) of volatility in some variable v at some program point: it denotes the expansion of the volatile bound for v due to the volatility in some preceding source expression S. It is thus a measure of the amount of "blame" to be assigned to S. Formally:

Definition 3. *Let v be a variable at some program point, and S be an expression. Let $[l, r]$ be v's volatile bound at that point, and $[\bar{l}, \bar{r}]$ be v's volatile bound at the same point but in a modified program where expression S has been stabilized to be computed under the strict evaluation model M_{str}. The* ***provenance pair*** *(Δ_l, Δ_r) for v and S is given by $\Delta_l = l - \bar{l}$, $\Delta_r = r - \bar{r}$.*

For example, when $\Delta_l < 0$, then the volatility in S causes v's left volatile bound to shift to the left—the bound interval expands. Likewise, when $\Delta_r > 0$, the volatility in S causes v's right volatile bound to shift to the right—the bound again expands. When $(\Delta_l, \Delta_r) = (0, 0)$, the volatility in S has no influence on v. This can be the case for example because S is not volatile, or is stable for the given input, or is not on any control path that reaches v's program point.

Precisely computing the provenance pair for a target variable v and a source expression S is expensive: applying Definition 3 would require the computation of volatile bounds from scratch as many times as we have volatile source expressions S. Since our goal is to employ our analysis at runtime (where small overhead is paramount), our method instead computes the necessary information in a linear sweep over the program. The price we pay is that we only compute an approximation (Δ'_l, Δ'_r), as described in the rest of this paper. The approximation has properties similar to those of (Δ_l, Δ_r): larger values $|\Delta'_l|$ and Δ'_r indicate heavier influence of S's volatility on v and hence suggest which source expression to stabilize first.

If the source expressions S_i on the path to the program point of v share program variables, stabilizing one of them generally affects the volatility contribution of the others. Our technique therefore proceeds in rounds, as illustrated in Sect. 2: after stabilizing one source expression, we recompute the (approximate) provenance pairs for v and all S_i from scratch, and repeat the ranking and statement stabilization process. For target variables v that are used in branches, assertions or other *decisions*, a natural point to stop is when the decision has become stable, i.e. platform independent, for all inputs. For other target variables, e.g. in branch-free programs, more heuristic stoppage criteria can be used; we discuss these further in our experiments in Sect. 8.

5 An Abstract Domain for Tracking Volatility

Similar in spirit to earlier work that tracks precision loss in a floating-point program relative to a real-arithmetic calculation [11,15], we use *affine arithmetic* to abstractly represent floating-point values under platform uncertainty. The

abstract values are affine-linear combinations of real numbers tagged with symbolic variables that label these real numbers as various "error contributions". Formally, the *provenance form* is a term

$$p = d + \sum_{l \in \mathfrak{L}} e_l \cdot \eta_l + e_* \cdot \eta_* \tag{7}$$

where d, e_l, and e_* are real numbers, $\mathfrak{L}$ is the set of program statement labels, and η_l and η_* are symbolic variables indicating reordering errors contributed by the statement at label l and *higher-order* error (combinations of different error terms after propagation), respectively. Value d represents the value of p under strict evaluation and incorporates rounding errors but not reordering uncertainty. We also define the real-valued *projection* of provenance form p as $\pi(p) = d + \sum_{l \in \mathfrak{L}} e_l + e_*$. The abstract values form a lattice via the provenance ordering defined as $p^i < p^j \Leftrightarrow \pi(p^i) < \pi(p^j)$. Binary arithmetic operations on forms p^i, p^j are defined as

$$p^i \pm p^j = (d^i \pm d^j) + \sum_{l \in \mathfrak{L}} (e_l^i \pm e_l^j) \cdot \eta_l + (e_*^i \pm e_*^j) \cdot \eta_* \tag{8}$$

$$p^i * p^j = (d^i * d^j) + \sum_{l \in \mathfrak{L}} (e_l^i * d^j + e_l^j * d^i) \cdot \eta_l + \left(\sum_{l_1, l_2 \in \mathfrak{L} \cup \{*\}} e_{l_1}^i * e_{l_2}^j \right) \cdot \eta_* \tag{9}$$

The multiplication rule in (9) is a simplification of the term obtained by multiplying out the sums for p^i and p^j: the higher-order terms have been merged into one, which would otherwise complicate the analysis significantly, with little benefit.

We use a pair $(L(v), U(v))$ of two provenance forms to abstractly representation the boundary points of volatile bound of some variable v. For $l \in \mathfrak{L}$, denote by $(\underline{e}_l, \bar{e}_l)$ the contribution of volatile errors of the statement at label l; then

$$L(v) = \underline{d} + \sum_{l \in \mathfrak{L}} \underline{e}_l \cdot \eta_l + \underline{e}_* \cdot \eta_* \,, \qquad U(v) = \bar{d} + \sum_{l \in \mathfrak{L}} \bar{e}_l \cdot \eta_l + \bar{e}_* \cdot \eta_* \,.$$

Forms $L(v)$ and $U(v)$ tell us which expressions affect v's volatile bound, in which direction (up or down), and by how much:

Example 1. *For the program shown in Sect. 2, the initial provenance forms for* `radiusSq` *are* $(0.015625, 0.015625)$. *All values* e_l *and* e_* *are 0, since there is no volatility in the input. Suppose after executing the statement in* `l2`*, we have* $(L(B), U(B)) = (-1129.18688965, -1129.186767578 + 0.00012207 \cdot \eta_{l2})$. *This means that the volatile error at* `l2` *increases the upper bound of* B *by* 0.00012207 *and has no effect on its lower bound.*

Our method is an instance of concretization-based abstract interpretation [6]. The collecting semantics for an abstract program state $(L(v), U(v))$ after assigning to variable v is the set of possible values of v under all evaluation models;

the concretization function thus is $\gamma((L(v), U(v))) = [\pi(L(v)), \pi(U(v))]$. With the concrete transfer function F for the program, our goal is an abstract transfer function G that maintains the following relationship:

$$F \circ \gamma \subseteq \gamma \circ G \ .$$

We construct G separately for volatile and non-volatile expressions (Sects. 6 and 7).

6 Abstract Transfer Functions for Volatile Expressions

If the assignment statement at label l contains the volatile expression $v = x_{11} \otimes x_{12} \ldots \otimes x_{1n} \oplus \ldots \oplus x_{m1} \otimes x_{m2} \ldots \otimes x_{mn}$, we need to propagate the volatile errors existing in x_{ij} (Sect. 6.1), but also calculate the new volatile error introduced at l in v by reordering and FMA (Sect. 6.2).

6.1 Propagating Existing Volatile Error

The propagation follows the spirit of interval analysis, but uses the operations for provenance forms from Sect. 5 and assumes strict evaluation in the current expression (new reordering error ignored). Given two (L, U) pairs x_i, x_j, we have

$$\begin{aligned}
x_i + x_j &= (L(x_i) + L(x_j) + \underline{d'}_{ij}, U(x_i) + U(x_j) + \bar{d'}_{ij}) \\
x_i - x_j &= (L(x_i) - U(x_j) + \underline{d'}_{ij}, U(x_i) - L(x_j) + \bar{d'}_{ij}) \\
x_i * x_j &= \\
&\quad (\min(L(x_i) * L(x_j), L(x_i) * U(x_j), U(x_i) * L(x_j), U(x_i) * U(x_j)) + \underline{d'}_{ij}, \\
&\quad \max(L(x_i) * L(x_j), L(x_i) * U(x_j), U(x_i) * L(x_j), U(x_i) * U(x_j)) + \bar{d'}_{ij})
\end{aligned}$$

The min/max functions use the order relation defined in Sect. 5. Values $\underline{d'}_{ij}/\bar{d'}_{ij}$ account for rounding errors of $\oplus, \ominus, \otimes$ and are defined as $d' = rd(\pi(p)) - \pi(p)$. Rounding error d' is added to p via $p + d' = (d + d') + \sum_{l \in \mathfrak{L}} e_l \cdot \eta_l + e_* \cdot \eta_*$. For example, in $L(x_i) + L(x_j) + \underline{d'}_{ij}$, $\underline{d'}_{ij}$ is defined as $\underline{d'}_{ij} = rd(\pi(L(x_i) + L(x_j))) - \pi(L(x_i) + L(x_j))$.

As in interval analysis, we ignore the relation between x_i and x_j; thus the resulting provenance form overapproximates possible values of v with M_{str}.

Theorem 4. *Let $(L'(v), U'(v))$ be the resulting provenance forms for v. Then:*

$$[\min_{I \in \mathbb{I}} v(I, M_{str}), \max_{I \in \mathbb{I}} v(I, M_{str})] \subseteq [\pi(L'(v)), \pi(U'(v))] \ ,$$

where $\mathbb{I} = [\pi(L(x_{11})), \pi(U(x_{11}))] \times \ldots \times [\pi(L(x_{mn})), \pi(U(x_{mn}))]$.

The theorem follows easily from the properties of interval analysis.

6.2 Calculating Fresh Volatile Error

By Formula (6) in Definition 2 of the volatile error, to calculate the volatile error introduced at l we need two bounds: the bound of v under M_{str}, which is approximated by $[\pi(L'(v)), \pi(U'(v))]$ (calculated in the previous step), and the volatile bound of v. In this section we show how to obtain a sound approximation for the latter. We only show this for the lower bound; the upper bound calculation is analogous.

Our approach consists of two steps. First we transform each sub-monomial $x_{i1} \otimes x_{i2} \ldots \otimes x_{in}$ into two-const form $c_i^{\dashv} \otimes c_i^{\vdash}$, done in Algorithm 6.1. The reason for choosing this form is that we need to consider possible ways of applying FMA between adjacent sub-monomials.

Algorithm 6.1. Compute the two-const form for monomial m

Input: $m := u_1 \otimes \ldots \otimes u_n$, where u_i can be a constant or a variable, $\downarrow u_i = \pi(L(u_i))$ and $\uparrow u_i = \pi(U(u_i))$

```
1  ↓m = +∞;
2  for (c_1, ..., c_n) ∈ {↓u_1, ↑u_1} × ... × {↓u_n, ↑u_n} do
3  |  (t^⊣, t^⊢) = getMin_mul(c_1, ..., c_n);
4  |  if ↓m > t^⊣ * t^⊢ then
5  |  |  ↓m = t^⊣ * t^⊢;
6  |  |  c^⊣ = t^⊣ ;
7  |  |  c^⊢ = t^⊢ ;
8  |  end
9  end
10 return (c^⊣, c^⊢);
```

Function $getMin_{mul}$ in Algorithm 6.1 is defined as

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

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

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

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 $N[i, j]$: the multiplication in FMA is done in real. We can prove that $(N[1, L[1, n]], N[L[1, n] + 1, n])$ is the pair such that its multiplication $N[1, L[1, n]] \otimes N[L[1, n] + 1, n]$ is the minimum value of the monomial.

After transformation, the whole polynomial expression is transformed into standard dot product: $c_1^{\dashv} \otimes c_1^{\vdash} \oplus \ldots \oplus c_n^{\dashv} \otimes c_n^{\vdash}$. In the second sub-step we obtain the lower bound of the dot product, $\downarrow v$, using a method presented in previous work [14, Sect. 2.4], which accounts for all possible evaluation models for the dot product expression. It can be shown that $[\downarrow v, \uparrow v]$ is an over-approximation of the

volatile bound of v. Together with $(L'(v), U'(v))$, we can now get the volatile error e_l for v of

$$\underline{e}_l = \downarrow v - \pi(L'(v)), \quad \bar{e}_l = \uparrow v - \pi(U'(v)) \ .$$

The final provenance form for v is $(L'(v) + \underline{e}_l \cdot \eta_l, U'(v) + \bar{e}_l \cdot \eta_l)$. It follows:

Theorem 5. *Let* $(L'(v)+\underline{e}_l \cdot \eta_l, U'(v)+\bar{e}_l \cdot \eta_l)$ *be the resulting provenance forms for* v*, and* $\mathbb{I} = [\pi(L(x_{11})), \pi(U(x_{11}))] \times \ldots \times [\pi(L(x_{mn})), \pi(U(x_{mn}))]$*. Then*

$$[\min_{I \in \mathbb{I}} \min_{M} v(I, M), \max_{I \in \mathbb{I}} \max_{M} v(I, M)] \subseteq [\pi(L'(v) + \underline{e}_l \cdot \eta_l), \pi(U'(v) + \bar{e}_l \cdot \eta_l)] \ .$$

6.3 Identifying the Cause of Volatility

To help the user fix reproducibility problems, we *classify* the cause of the volatility into three categories: *FMA*, *Order* and *FMA+Order*, which respectively indicates that the volatility is due to the use/non-use of FMA, reordering of the computation, or both. The definitions of the three categories are shown in Table 4. Here the $[\downarrow v_{nofma}, \uparrow v_{nofma}]$ represents the volatile bound of v without considering FMA contraction. It can be obtained by modifying the return value in Algorithm 6.1. Instead of returning the tuple $(c^{\dashv}, c^{\vdash})$, we simply return the floating-point value $c^{\dashv} \otimes c^{\vdash}$. Then the whole volatile expression is transformed to $v = c_1 \otimes 1 \oplus \ldots \oplus c_n \otimes 1$, where $c_i = c_i^{\dashv} \otimes c_i^{\vdash}$. The second sub-step is the same as in Sect. 6.2.

Table 4. Categories of volatility

Category	Definition
Stable	$\downarrow v = \uparrow v$
FMA	$\downarrow v \neq \uparrow v$ $\wedge \downarrow v_{nofma} = \uparrow v_{nofma}$
Order	$\downarrow v \neq \uparrow v$ $\wedge \downarrow v_{nofma} = \downarrow v$ $\wedge \uparrow v_{nofma} = \uparrow v$
FMA+Order	otherwise

7 Transfer Functions for Non-volatile Expressions

Numerical programs generally contain expressions other than polynomials, such as involving division and *sqrt* operations. We assume that such expressions behave the same on all platforms and hence do not introduce new volatile error. We still need to propagate existing volatile error through them, which is the topic of this section. Our approach applies to any uni-variate function that is *monotone and twice continuously differentiable* in its domain.

Let $\vec{\eta} = (\eta_i)$ be an $|\mathfrak{L}|$-dimensional vector. The provenance form $p = d + \sum_{l \in \mathfrak{L}} e_l \cdot \eta_l + e_* \cdot \eta_*$ can be viewed as a function f of $|\mathfrak{L}| + 1$ variables, namely all η_i and η_*, defined over the line segment $\overline{AB}$ from point $A = (\vec{0}, 0)$ to point $B = (\vec{1}, 1)$: note that $f(\vec{1}, 1) = \pi(p)$. Let g be a uni-variate function such that $\varphi = g \circ f$ is twice continuously differentiable in $\overline{AB}$. By the Taylor expansion theory [17], there exists a point C in the interior of $\overline{AB}$ such that

$$\varphi(B) = \varphi(A) + \sum_{l \in \mathfrak{L} \cup \{*\}} \frac{\partial \varphi}{\partial \eta_l}(A) \cdot 1 + \frac{1}{2} \sum_{l_1, l_2 \in \mathfrak{L} \cup \{*\}} \frac{\partial^2 \varphi}{\partial \eta_{l_1} \partial \eta_{l_2}}(C) \cdot 1 \cdot 1. \qquad (10)$$

Now our plan is to transfer Formula (10) back to provenance form. Recall that, in our definition of provenance form, e_l reflects the shift of the volatile bound (lower or upper) due to the statement at l. Thus, by the definition of derivative, $\frac{\partial \varphi}{\partial \eta_l}(A)$ approximates the change of the volatile bound of g because of e_l. We also need to handle the formula's final term, $\frac{1}{2}\sum_{l_1,l_2\in\mathfrak{L}\cup\{*\}} \frac{\partial^2 f}{\partial \eta_{l_1}\partial \eta_{l_2}}(C)$, which contains the unknown parameter C. Here we apply interval analysis on the values along $\overline{AB}$ that C can take, to get an interval $\boldsymbol{e''}$. This interval overapproximates the change of the bound of g due to higher-order error terms according to the definition of second derivative. Based on the above discussion, we get the following "quasi-provenance form" for g for the given input p.

$$g(p) = \varphi(A) + \sum_{l\in\mathfrak{L}} \frac{\partial \varphi}{\partial \eta_l}(A)\cdot\eta_l + (\frac{\partial \varphi}{\partial \eta_*}(A) + \boldsymbol{e''})\cdot\eta_* \tag{11}$$

The only difference to the provenance form is that the coefficient of η_* is an interval instead of a constant. We also define $[\downarrow g(p), \uparrow g(p)]$ as

$$\downarrow g(p) = \varphi(A) + \sum_{l\in\mathfrak{L}} \frac{\partial \varphi}{\partial \eta_l}(A)\cdot\eta_l + \downarrow(\frac{\partial \varphi}{\partial \eta_*}(A) + \boldsymbol{e''})\cdot\eta_*$$

$$\uparrow g(p) = \varphi(A) + \sum_{l\in\mathfrak{L}} \frac{\partial \varphi}{\partial \eta_l}(A)\cdot\eta_l + \uparrow(\frac{\partial \varphi}{\partial \eta_*}(A) + \boldsymbol{e''})\cdot\eta_*$$

Using the quasi-provenance form, we can design the volatile error propagation for g as the follows. If g is monotonously increasing in interval $[\pi(L(v)), \pi(U(v))]$, we have $g((L(v), U(v))) = (\downarrow g(L(v)), \uparrow g(U(v)))$. If g is monotonously decreasing, we have $g((L(v), U(v))) = (\downarrow g(U(v)), \uparrow g(L(v)))$.

Theorem 6. *Let g be a uni-variate, monotone, and twice continuously differentiable function. Given abstract input $(L(v), U(v))$, let $(L'(v), U'(v))$ be the abstract result (obtained via the abstract transfer function of g). Then*

$$[g(\pi(L(v)), g(\pi(U(v)))] \subseteq [\pi(L'(v)), \pi(U'(v))] \ .$$

The above discussion assumes that g can be calculated in infinite precision. In floating-point reality we need to consider rounding errors for g. This can be accommodated by attaching correction terms to the resulting provenance forms: $(L'(v)+\underline{d}, U'(v)+\bar{d})$, where $\underline{d} = \widetilde{g}(\pi(L(v)))-\pi(L'(v))$, $\bar{d} = \widetilde{g}(\pi(U(v)))-\pi(U'(v))$, and $\widetilde{g}$ is the floating-point version of g.

8 Implementation and Evaluation

We have implemented the above techniques in a runtime library. The core of the library is a customized datatype called `ifloat`, which keeps track of the volatile errors during execution, for each variable. Our library can be applied to programs

that do not use mixed floating-point types or mixed rounding modes. In our experiments we use single-precision float as the numeric data type, and *round-to-nearest-ties-to even* as the rounding mode (as in most programs). Polynomial expressions containing parentheses have "partial volatility"; we treat them by moving the parenthesized part outside the expression, via a temporary variable.

The calculations of provenance forms are defined in $\mathbb{R}$, as shown in Sect. 5. In our implementation we use the `mpq_class` type in the GMP library [13] as an approximation. We have overloaded the following operators in our `ifloat` datatype: $+ \; - \; * \; /$ `sqrt`. Note that the result of `sqrt` may be irrational, in which case Formula (10) may not be representable by `mpq_class`. To solve this problem, we use a double precision float together with outward rounding modes to get an interval that contains the true value of `sqrt`. As a result, we get an *interval linear form* [18] of Formula (11),

$$\sqrt{p} = [\sqrt{d}] + \sum_{l \in \mathfrak{L}} \frac{e_l}{2[\sqrt{d}]} \cdot \eta_l + (\frac{e_*}{2[\sqrt{d}]} + \boldsymbol{e''}) \cdot \eta_*,$$

where $[\sqrt{d}] = [RD(\sqrt{d}), RU(\sqrt{d})]$ (rounding down and up, resp.). Consequently,

$$\downarrow\sqrt{p} = \downarrow[\sqrt{d}] + \sum_{l \in \mathfrak{L}} \downarrow\frac{e_l}{2[\sqrt{d}]} \cdot \eta_l + \downarrow(\frac{e_*}{2[\sqrt{d}]} + \boldsymbol{e''}) \cdot \eta_*$$

$$\uparrow\sqrt{p} = \uparrow[\sqrt{d}] + \sum_{l \in \mathfrak{L}} \uparrow\frac{e_l}{2[\sqrt{d}]} \cdot \eta_l + \uparrow(\frac{e_*}{2[\sqrt{d}]} + \boldsymbol{e''}) \cdot \eta_*.$$

It can be shown that Theorem 6 still holds.

Library usage. Our library can be used in the classical test-evaluation-fix iterative fashion. Users replace all native float types with `ifloat` and label all assignment statements that contain volatile expressions. Currently we make these changes manually; they can easily be automated. Our library outputs the provenance forms for user-selected variables. From the two forms the user can locate which statement makes the most significant contribution to the target variable's volatility, and how to stabilize it. After (partial) stabilization, we re-run the analysis. If the resulting bound is within a user-specified threshold, we can guarantee that the actual volatility will not exceed the same threshold, since our volatile bound is a conservative approximation.

Benchmarks. We have tested our approach on a number of numeric programs. Benchmarks *fft* (Fast Fourier Transform) and *sor* (Jacobi Successive Over-relaxation) are from SciMark 2.0 [19]; *nbody* [9] models the orbits of Jovian planets. The remaining programs are from a numerical analysis book [4]: *triple* is the Gaussian triple integral algorithm; *adam* is the adams-forth order predictor-corrector algorithm; *crout* is the crout reduction for tri-diagonal linear systems; *choleski* and *ldl* are standard algorithms that factor a positive-definite matrix. The benchmarks can be found at http://github.com/yijiagu/ifloat.

Experiments. We have tested our benchmarks with 10 random generated inputs (except *nbody*, which comes with its own test inputs). All experiments are run on Ubuntu 15.10 with 8 GB memory; Table 5 shows the results. Column **Input** specifies the input matrix size; $*$ means that the benchmark requires scalar 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. These volatile bounds are sound based on the theorems in Sect. 6 and 7. If the output is a matrix as well, 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**.

Table 5. Volatility for the benchmarks

Program	Input	Variable	Volatile bound	v_{str}
sor	[100 × 100]	G[67][55]	(0.720786273, 0.720786631)	0.720786452
fft	[16 × 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 × 10]	A[0]	(1.282013535, 1.282219410)	1.282117963
choleski	[15 × 15]	A[11][11]	(4.187705517, 4.187705994)	4.187705994
ldl	[15 × 15]	A[11][10]	(0.102230683, 0.102230750)	0.102230720

Table 6 shows the provenance information for the selected variable. We only list the labels whose statements contribute the most to the target variable's lower (Column 2) and upper bound (Column 4). Table 6 also compares the contribution calculated by our approach (Columns 2+4) to the precise shift, according to Definition 3 (Column 3+5). In most cases, these two sets of values are very close.[2] It shows that the provenance outputs from our library are indeed a good approximation of statements' volatility contribution to the final result. Noted that in some cases the actual value is larger than the predicted value. However, this does not violate our claim that the volatile bound is sound. In all experiments, our approach accurately pinpoints the statements that contribute the most to the volatility of the final result. Thus, instead of naively recompiling and rerunning the analysis for each target variable and source expression, library users execute the program with our library *once*. From the output provenance forms they can identify the most-to-blame statements and to what extent they may improve the situation by stabilizing these statements. To assist in this process, Columns 2+4 also list the cause of statements' volatility.

[2] An exception is $A[11][11]$ for *choleski*: inspection shows that this anomaly is due to the rounding error. In fact, its left bound 4.187705517 and right bound 4.187705994 are two adjacent floating-point numbers.

Table 6. Provenance information for the benchmarks

Variable	Predicted ε_{max}	Actual ε_{max}	Predicted $\bar{e}_{max}$	Actual $\bar{e}_{max}$
G[67][55]	L1: −0.000000132 (Reorder)	−0.000000179	L1: 0.000000178 (Reorder)	0.000000179
X[14]	L2: −0.000000158 (FMA+Reorder)	−0.000000179	L2: 0.000000079 (FMA+Reorder)	0.000000060
energy	L7: −0.000000018 (FMA+Reorder)	−0.000000029	L9: 0.000000014 (FMA+Reorder)	0.000000000
AJ	L2: −0.000004037 (FMA)	−0.000003815	L8: 0.000005997 (FMA)	0.000007629
W0	L3: −0.000000014 (FMA+Reorder)	0.000000000	L2: 0.000000016 (FMA+Reorder)	0.000000060
A[0]	L1: −0.000098395 (FMA)	−0.000100494	L1: 0.000098380 (FMA)	0.000101450
A[11][11]	L1: −0.000000007 (FMA+Reorder)	−0.000000477	L2: 0.000000027 (FMA+Reorder)	0.000000000
A[11][10]	L2: −0.000000040 (FMA+Reorder)	−0.000000037	L2: 0.000000035 (FMA+Reorder)	0.000000030

The code linked against our library is currently up to 3 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 arithmetic in the process. We point out that the performance of the library is not our main concern: we view it as a unit testing tool for examining a program's numerically intensive parts. Our solution is effective in that it eases the testing workload by avoiding multiple recompilations and reruns on different platforms. The runtime in our experiments is acceptable; each test input takes less than a few seconds. In the future, we plan to replace rational arithmetic in the analysis by floating-point calculations, to enable application to larger examples.

9 Related Work

Analyzing the behavior of numerical programs across computing environments has become an important research topic, due to an increased awareness of reproducibility issues on heterogeneous (CPU/GPU/FPGA) platforms. The research presented here was inspired by our own work in [14], where we designed an efficient technique to compute the volatile bound of an expression, for a fixed input. In the present work, we embed this technique in a dynamic analysis framework, and go a significant step further: tracing volatility errors back to relevant source statements and specific causes (FMA/reordering), and stabilizing the program, by partially fixing the evaluation of these source statements.

[2] presents a formally verified `C` compiler that guarantees IEEE-compliant floating-point machine code, which is achieved by enforcing "a single way to

compile" [2], 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, nor how much stability is "healthy" for the program. Work in [3,21] proves a maximum rounding error *under platform variations*. The approach is deductive and not suitable for identifying inputs that cause platform dependence. Since our error representation (the provenance form) allows us to isolate errors due to volatility, we are able to trace sources of platform sensitivity, and can offer ways to repair it; a question that has not been addressed in any previous work, to the best of our knowledge.

The methodology used in our work is in part inspired by research on tracing the rounding error propagation for numerical programs. Fluctuat [10–12] is a static analysis tool based on abstract interpretation that locates the sources of rounding errors in the program. [7] designs a runtime library that provides a guaranteed upper bound of rounding errors for programs written in Scala. All these works use affine arithmetic [20] as the underlying algebraic structure to keep track of the rounding error. In this paper, we adopt a similar data structure, manifest in the provenance form, to instead trace the volatile error.

10 Conclusions and Future Work

In this paper we have established that platform and compiler dependencies of numeric code can be traced back to their sources dynamically, incurring a performance penalty that is acceptable for software test runs. The slow-down is currently too large to permit running the analysis as monitors in deployed code. Future work includes decreasing this performance hit, for example by conservatively using floating-point calculations during the analysis, rather than expensive but more precise rational arithmetic.

References

1. IEEE Standards Association. IEEE standard for floating-point arithmetic (2008). http://grouper.ieee.org/groups/754/
2. Boldo, S., Jourdan, J.-H., Leroy, X., Melquiond, G.: A formally-verified C compiler supporting floating-point arithmetic. In: 21st IEEE Symposium on Computer Arithmetic (ARITH), pp. 107–115. IEEE (2013)
3. Boldo, S., Nguyen, T.M.T.: Hardware-independent proofs of numerical programs. In: Muñoz, C. (ed.) Second NASA Formal Methods Symposium (NFM 2010), vol. NASA/CP-2010-216215, pp. 14–23. NASA, Washington D.C., April 2010
4. Burden, R.L., Faires, J.D.: Numerical analysis. Cengage Learning (2010)
5. Corden, M.J., Kreitzer, D.: Consistency of floating-point results using the Intel® compiler or why doesn't my application always give the same answer? (2010). http://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979, pp. 269–282. ACM, New York (1979)

7. Darulova, E., Kuncak, V.: Trustworthy numerical computation in Scala. ACM Sigplan Not. **46**, 325–344 (2011). ACM
8. de Dinechin, F.: Computing with floating point. http://lyoncalcul.univ-lyon1.fr/IMG/pdf/FloatingPoint.pdf
9. Fulgham, B., Gouy, I.: The Computer Language Benchmarks Game (2010). http://shootout.alioth.debian.org
10. Putot, S., Goubault, E., Martel, M.: Static analysis-based validation of floating-point computations. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Num. Software with Result Verification. LNCS, vol. 2991, pp. 306–313. Springer, Berlin (2004). doi:10.1007/978-3-540-24738-8_18
11. Goubault, E.: Static analyses of the precision of floating-point operations. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 234–259. Springer, Berlin (2001). doi:10.1007/3-540-47764-0_14
12. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Berlin (2011). doi:10.1007/978-3-642-18275-4_17
13. Granlund, T., and the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library, 6.1.1 edn. (2016). http://gmplib.org/
14. Gu, Y., Wahl, T., Bayati, M., Leeser, M.: Behavioral non-portability in scientific numeric computing. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 558–569. Springer, Berlin (2015). doi:10.1007/978-3-662-48096-0_43
15. Martel, M., Cea Recherche Technologique: Semantics of roundoff error propagation in finite precision computations. J. High. Order Symbolic Comput. **19**, 7–30 (2006)
16. Meng, Q., Humphrey, A., Schmidt, J., Berzins, M.: Preliminary experiences with the Uintah framework on Intel Xeon Phi and stampede. In: Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery, p. 48. ACM (2013)
17. Mikusinski, P., Taylor, M.: An Introduction to Multivariable Analysis from Vector to Manifold. Springer Science & Business Media, New York (2012)
18. Miné, A.: The octagon abstract domain. High. Order Symbol. Comput. **19**(1), 31–100 (2006)
19. Pozo, R., Miller, B.: SciMark 2.0, December 2002. http://math.nist.gov/scimark2/
20. Stolfi, J., De Figueiredo, L.H.: An introduction to affine arithmetic. Trends Appl. Comput. Math. **4**(3), 297–312 (2003)
21. Tuyen, N.T.M., Marché, C.: Proving floating-point numerical programs by analysis of their assembly code. Research Report RR-7655, INRIA, June 2011