Finding Root Causes of Floating Point Error

Alex Sanchez-Stern
University of California San Diego
United States of America
alexss@eng.ucsd.edu

Pavel Panchekha
University of Washington
United States of America
pavpan@cs.washington.edu

Sorin Lerner
University of California San Diego
United States of America
lerner@cs.ucsd.edu

Zachary Tatlock
University of Washington
United States of America
ztatlock@cs.washington.edu

Abstract

Floating-point arithmetic plays a central role in science, engineering, and finance by enabling developers to approximate real arithmetic. To address numerical issues in large floating-point applications, developers must identify root causes, which is difficult because floating-point errors are generally non-local, non-compositional, and non-uniform.

This paper presents Herbgrind, a tool to help developers identify and address root causes in numerical code written in low-level languages like C/C++ and Fortran. Herbgrind dynamically tracks dependencies between operations and program outputs to avoid false positives and abstracts erroneous computations to simplified program fragments whose improvement can reduce output error. We perform several case studies applying Herbgrind to large, expert-crafted numerical programs and show that it scales to applications spanning hundreds of thousands of lines, correctly handling the low-level details of modern floating point hardware and mathematical libraries and tracking error across function boundaries and through the heap.

CCS Concepts • **Software and its engineering** → *Software maintenance tools*;

Keywords floating point, debugging, dynamic analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192411

ACM Reference Format:
Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192411

1 Introduction

Large floating-point applications play a central role in science, engineering, and finance by enabling engineers to approximate real number computations. Ensuring that these applications provide accurate results (close to the ideal real number answer) has been a challenge for decades [17–21]. Inaccuracy due to rounding errors has led to market distortions [25, 31], retracted scientific articles [1, 2], and incorrect election results [38].

Floating-point errors are typically silent: even when a grievous error has invalidated a computation, it will still produce a result without any indication things have gone awry. Recent work [3, 4] has developed dynamic analyses that detect assembly-level operations with large intermediate rounding errors, and recent static error analysis tools have been developed to verify the accuracy of small numerical kernels [11, 34].

However, after floating-point error has been detected, there are no tools to help a developer diagnose and debug its *root cause*. The root cause is the part of a computation whose improvement would reduce error in the program's outputs. Previous work in the area [4] has called out root cause analysis as a significant open problem that needs to be addressed. This paper addresses this problem with a tool that works on large, real, expert-crafted floating-point applications written in C/C++, and Fortran.

In practical settings, root causes are difficult to identify precisely. Root causes often involves computations that cross function boundaries, use the heap, or depend on particular inputs. As such, even though root causes are part of the computation, they rarely appear as delineable syntactic entities in the original program text. The key challenge, then, is identifying root causes and suitably abstracting them to a form

that enables numerical analysis and facilitates improving the accuracy of the program.

From a developer's perspective, identifying and debugging numerical issues is difficult for several reasons. First, floating-point error is non-compositional: large intermediate errors may not impact program outputs and small intermediate errors may blow up in a single operation (due to, e.g., cancellation or overflow). Second, floating-point errors are often non-local: the source of an error can be far from where it is observed and may involve values that cross function boundaries and flow through heap-allocated data structures. Third, floating-point errors are non-uniform: a program may have high error for certain inputs despite low or non-existent error on other inputs.

This paper presents Herbgrind, a dynamic binary analysis that identifies *candidate root causes* for numerical error in large floating-point applications. First, to address the non-compositionality of error, Herbgrind records operations with intermediate error and tracks the *influence* of these operations on program outputs and control flow. Second, to address the non-locality of error, Herbgrind provides symbolic expressions to describe erroneous computation, abstracting the sequence of operations that cause error to program fragments which facilitate numerical analysis. Finally, to address the non-uniformity of error, Herbgrind characterizes the inputs to erroneous computations observed during analysis, including the full range as well as the subset that caused significant error.

We demonstrate Herbgrind's effectiveness by identifying the root causes of error in three expert-written numerical applications and benchmarks. We find that Herbgrind handles the tricky floating-point manipulations found in expert-written numerical code, and can identify real sources of floating-point error missed by experts when writing important software. To further characterize the impact of Herbgrind's key components, we carried out a set of smaller experiments with the FPBench floating-point benchmark suite [10], and find that each of Herbgrind's components is crucial to its accuracy and performance.

To the best of our knowledge, Herbgrind provides the first approach to identifying and summarizing root causes of error in large numerical programs. Herbgrind is implemented in the Valgrind binary instrumentation framework, and achieves acceptable performance using several key optimizations.[1] Building Herbgrind required developing the following contributions:

- An analysis that identifies candidate root causes by tracking dependencies between sources of error and program outputs, abstracting the responsible computations to an improvable program fragment, and characterizing the inputs to these computations (Section 4).
- An implementation of this analysis that supports numerical code written in low-level languages like C/C++ and Fortran and handles the complexities of modern floating point hardware and libraries (Section 5).
- Key design decisions and optimizations required for this implementation to achieve acceptable performance when scaling up to applications spanning hundreds of thousands of lines (Section 6).
- An evaluation of Herbgrind including bugs found "in the wild" and measurements of the impact of its various subsystems (Section 7 and Section 8).

2 Background

A floating-point number represents a real number of the form $\pm(1 + m)2^e$, where m is a fixed-point value between 0 and 1 and e is a signed integer; several other values, including two zeros, two infinities, not-a-number error values, and subnormal values, can also be represented. In double-precision floating point, m is a 52-bit value, and e is an 11-bit value, which together with a sign bit makes 64 bits. Simple operations on floating-point numbers, such as addition and multiplication, are supported in hardware on most computers.

2.1 Floating-Point Challenges

Non-compositional error Individual floating-point instructions are always evaluated as accurately as possible, but since not all real numbers are represented by a floating-point value, some error is necessarily produced. Thus, the floating-point sum of x and y corresponds to the real number $x+y+(x+y)\epsilon$, where ϵ is some small value induced by rounding error.[2] Floating-point operations implemented in libraries also tend to bound error to a few units in the last place (ulps) for each operation, but are generally not guaranteed to be the closest result. However error grows when multiple operations are performed. For example, consider the expression $(x+1)-x = 1$. The addition introduces error ϵ_1 and produces $x + 1 + (x + 1)\epsilon_1$. The subtraction then introduces ϵ_2 and produces

$$1 + (x + 1)\epsilon_1 + \epsilon_2 + (x + 1)\epsilon_1\epsilon_2.$$

Since x can be arbitrarily large, the $(x+1)\epsilon_1$ term can be large; in fact, for values of x on the order of 10^{16}, the expression $(x+1)-x$ evaluates to 0, not 1. The influence of intermediate errors on program output can be subtle; not only can accurate intermediate operations compute an inaccurate result, but intermediate error can cancel to produce an accurate result. Experts often orchestrate such cancellation, which poses a challenge to dynamic analysis tools trying to minimize false positives.

[1] The implementation of Herbgrind is publicly available at https://github.com/uwplse/herbgrind

[2] For cases involving subnormal numbers, the actual error formula is more complex; these details are elided here.

Non-local error Floating-point error can also be non-local; the cause of a floating point error can span functions and thread through data structures. Consider the snippet:

```
double foo(struct Point a, struct Point b) {
  return ((a.x + a.y) - (b.x + b.y)) * a.x;
}
double bar(double x, double y, double z) {
  return foo(mkPoint(x, y), mkPoint(x, z));
}
```

The foo and bar functions individually appear accurate. However, bar's use of foo causes inaccuracy. For example for inputs x=1e16, y=1, z=0, the correct output of bar is 1e16, yet bar instead computes 0. However, this combination of foo and bar can be computed more accurately, with the expression $(y - z) \cdot x$. Note that in this example, understanding the error requires both reasoning across function boundaries and through data structures.

Non-uniform error For a given computation, different inputs can cause vastly different amounts of floating-point error. Effectively debugging a numerical issue requires characterizing the inputs which lead to the root cause impacting output accuracy. For example, consider the snippet:

```
double baz(double x){
  double z =  1 / (x - 113);
  return (z + M_PI) - z;
}
```

When debugging baz, it's important to know what inputs baz is called on in the context of the larger program. For most inputs, baz is accurate; if baz is only called on inputs far from 113, then a programmer need not consider it problematic. However, for values of x near 113, baz suffers significant rounding error, because z becomes very large, and then most of the bits of π are lost to catastrophic cancellation. To diagnose the root cause of error in a program containing baz, programmers need to know whether baz is called on inputs near 113; if not, they may waste time investigating baz's behavior on inputs near 113 when those inputs are never seen in practice. In this example, understanding the error requires reasoning about the inputs on which the fragment of code will be executed.

2.2 Existing Debugging Tools

Table 1 compares the tools most closely related to Herbgrind.

Error detection All of the tools compared are dynamic analyses which attempt to detect error. Like Herbgrind, FpDebug[4] uses high-precision shadow values to track the error of floating-point computations; Verrou[12] and BZ[3] use heuristic methods to detect possible instances of error.

While individual hardware floating-point operations are accurate, more complex operations, like trigonometric functions, are generally implemented in low-level math libraries which make use of hundreds of floating-point instructions and bit manipulations for each operation, and are painstakingly crafted by experts [16, 28]. Even higher level operations, like those on matrices and vectors, are implemented in thousands of lower-level operations, often building on both hardware floating-point instructions and lower-level math libraries [22]. Previous tools report error locations as individual opcode addresses, which may be deep within the internals of the implementation of a sophisticated operation. This is unfortunate, as most users are unwilling and/or unable to modify a `libm` or BLAS implementation; instead such operations should be treated as atomic black boxes so that users can focus on ensuring such operations are accurately used at a higher level of abstraction. In contrast, Herbgrind supports abstracting over such library calls which enables more useful error location reporting and permits meaningful shadowing in high-precision.

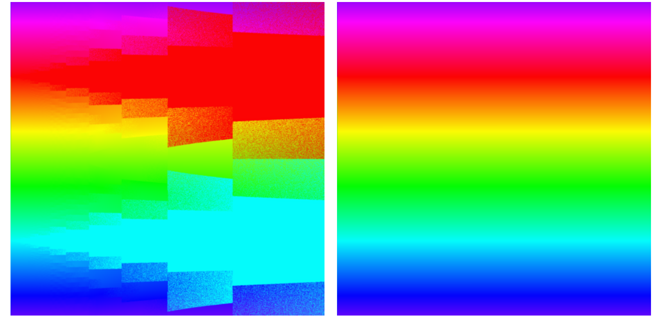

Figure 1. Complex plotter output before (left) and after (right) diagnosing and fixing a floating-point error.

Root cause analysis There are two ways in which floating-point error can affect observable program behavior: either by flowing directly to an output, or changing control flow. BZ [3] can detect changes in control flow due to error; but cannot reason about how error affects program outputs. Verrou and FpDebug have no support for detecting observable program behavior affected by error. When an error is detected, FpDebug reports the binary address where the error was found, while Verrou and BZ only report that something has gone wrong, not where it has gone wrong. In contrast, Herbgrind characterizes both the full range and error-inducing inputs to the problematic computations observed during analysis.

3 Overview

We illustrate Herbgrind by detailing its use on a simple complex function plotter. Given function $f : \mathbb{C} \to \mathbb{C}$, region $R = [x_0, x_1] \times [y_0, y_1]$ in the complex plane, and a resolution, the plotter tiles R with a grid of pixels, and colors each pixel based on $\arg(f(x + iy))$ at the center (x, y) of each pixel.[3] Since the output of the program is an image, minor errors in

[3] arg is also known as `atan2`

Table 1. Comparison of floating-point error detection tools. Note that all tools are run on distinct benchmark suites.

Feature	FpDebug	BZ	Verrou	Herbgrind
Error Detection				
Dynamic	✓	✓	✓	✓
Detects Error	✓	✓	✓	✓
Shadow Reals	✓	✗	✗	✓
Local Error	✗	✗	✗	✓
Library Abstraction	✗	✗	✗	✓
Root Cause Analysis				
Output-Sensitive Error Report	✗	✗	✗	✓
Detect Control Divergence	✗	✓	✗	✓
Localization	Opcode Address	None	None	Abstracted Code Fragment
Characterize Inputs	✗	✗	✗	✓
Other				
Automatically Re-run in High Precision	✗	✓	✗	✗
Overhead*	395x	7.91x	7x	574x

the evaluation of f can usually be ignored. However, floating-point error can compound in unexpected ways. Consider the function[4]

$$f(z) = 1/\left(\sqrt{\Re(z)} - \sqrt{\Re(z) + i\exp(-20z)}\right).$$

To evaluate this function, the plotter must provide codes for evaluating the square root of a complex number. The standard formula is

$$\sqrt{x+iy} = \left(\sqrt{\sqrt{x^2+y^2}+x} + i\sqrt{\sqrt{x^2+y^2}-x}\right)/\sqrt{2},$$

where the square roots in the definitions are all square roots of real numbers (typically provided by standard math libraries).

Implementing f using this formula and plotting the region $R = [0, \frac{1}{4}] \times [-3, 3]$ results in the left image of Figure 1. The speckling is not an accurate representation of f; in fact, f is continuous in both x and y throughout R.

Herbgrind uses three key components (detailed in Section 4) to identify the root cause of error in the plotting program: (1) a shadow taint analysis, which tracks the ways that erroneous operations influence important program locations called *spots*; (2) a shadow symbolic execution, which builds expressions representing the computations that produced each value; and (3) an input characterization system, which maintains information about the set of inputs to each computation.

Herbgrind detects that the plotter computes wrong pixel values due to significant error from a subtraction with high local error:

```
Compare @ main.cpp:24 in run(int, int)
231878 incorrect values of 477000
Influenced by erroneous expressions:
  (FPCore (x y)
    :pre (and (<= -2.061152e-9 x 2.497500e-1)
              (<= -2.619433e-9 y 2.645912e-9))
        (- (sqrt (+ (* x x) (* y y))) x))
 Example problematic input: (2.061152e-9, -2.480955e-12)
```

This report shows that at line 24 of the `main.cpp` source file, inaccuracy is caused by the expression:

$$\sqrt{x^2+y^2} - x.$$

Running Herbie [29] on the above expression produces this more accurate version.

$$\sqrt{x^2+y^2} - x \rightsquigarrow \begin{cases} \sqrt{x^2+y^2} - x & \textbf{if } x \le 0 \\ y^2/\left(\sqrt{x^2+y^2}+x\right) & \textbf{if } x > 0 \end{cases}.$$

Substituting this expression back into the original complex square root definition (and simplifying) yields

$$\sqrt{x+iy} = \frac{1}{\sqrt{2}}\begin{cases} |y|/\sqrt{\sqrt{x^2+y^2}-x} + i\sqrt{\sqrt{x^2+y^2}-x} & \textbf{if } x \le 0 \\ \sqrt{\sqrt{x^2+y^2}+x} + i|y|/\sqrt{\sqrt{x^2+y^2}+x} & \textbf{if } x > 0 \end{cases}$$

for the complex square root. Replacing the complex square root computation in the plotter with this alternative implementation fixes the inaccurate computation, as confirmed by running Herbgrind on the repaired program[5]. The fixed code produces the right graph in Figure 1.

While error in the complex plotter is primarily in a single compilation unit, real world numerical software often has many numerical components which interact, and root causes often cannot be isolated to one component. We evaluate

[4] $\Re(z)$ indicates the real (non-imaginary) part of z

[5] Automating the process of inserting improved code back into the binary is left to future work, but Herbgrind can provide source locations for each node in the extracted expression.

Herbgrind on several large numerical systems which exhibit these issues in Section 7.

4 Analysis

The implementation of Herbgrind's analysis requires handling complex machine details including different value precisions, multiple distinct storage types, and bit-level operations, which we detail in Section 5. To first clarify the core concepts, this section describes Herbgrind's analysis in terms of an abstract float machine. Herbgrind's analysis consists of three components: a spots-and-influences system to determine which operations influence which program outputs (Section 4.2), a symbolic expression system to track computations across function and heap data structure boundaries (Section 4.3), and an input characteristics system to determine on which inputs the computation is erroneous or accurate (Section 4.4).

4.1 Abstract Machine Semantics

The abstract machine has floating-point values and operations as well as memory and control flow (Figure 2). A machine contains mutable memory $\mathcal{M} : \mathbb{Z} \rightarrow (\mathbb{F} \mid \mathbb{Z})$ which stores floating-point values or integers and a program counter $\mathrm{pc} : \mathbb{Z}$ that indexes into the list of program statements. Statements include: computations, control operations, and outputs. A program is run by initializing the memory and program counter, and then running statements until the program counter becomes negative.

Herbgrind's analysis describes candidate root causes for programs on this abstract machine (Figure 3 and Figure 4) by updating analysis information for each instruction in the program. Each computation instruction with floating-point output is associated with an operation entry, which describes the computation that led up to that operation, and a summary of the values that computation takes. All other instructions have a spot entry, which lists the error at that location and the erroneous computations that influence the location (Section 4.2).

Shadow Reals Floating-point errors are generally silent: even when error invalidates a computation, that computation still produces a floating-point value without any indication that the value is erroneous. Herbgrind follows prior work in detecting the floating-point errors in a program by computing a real number shadow for every floating-point value in ordinary program memory.[6] For each such statement, Herbgrind executes the statement's operation in the reals on real-number shadow inputs, and stores the resulting real number to the real-number shadow memory. Herbgrind's handling of mathematical libraries and similar details is discussed in Section 5.

[6]While the abstract analysis is defined in terms of computing over reals, the implementation must settle merely for high precision (e.g., 1000-bit mantissa) approximations (Section 5.1).

4.2 Spots and Influence Shadows

Herbgrind uses the real-number execution to measure the floating-point error at program outputs, conditional branches, and conversions from floating-point values to integers; these three types of program locations are collectively called *spots*. Since error is non-compositional, the root cause of error at a spot can be far from the spot itself. To overcome this, Herbgrind identifies candidate root causes and tracks their influence on spots using a taint analysis: every floating-point value has a "taint" set of influencing instructions which is propagated by computation.

Herbgrind uses *local error* to determine which floating-point operations cause error (Figure 4). Local error [29] measures the error an operation's output would have even if its inputs were accurately computed, and then rounded to native floats. Using local error to assess operations avoids blaming innocent operations for erroneous operands. Any operation whose local error passes a threshold T_ℓ is treated as a candidate root cause.[7]

In the case of the complex plotter from Section 3, the subtraction at the top of the reported expression was determined to have high local error, resulting in it being tracked.

4.3 Symbolic Expressions

Identifying the operation that introduced error, as the influences system does, is not sufficient to understand the error, because floating-point error is non-local: to understand why an operation is erroneous requires understanding how its inputs were computed. In many cases those inputs are separated from the erroneous operation by function boundaries and heap data structure manipulations. Herbgrind analyzes through those boundaries by providing *symbolic expressions* for the erroneous operation and its inputs.

Symbolic expressions represent an abstract computation that contains a candidate root cause. Each symbolic expression contains only floating-point operations: it therefore abstracts away function boundaries and heap data structures. A symbolic expression must be general enough to encompass any encountered instance of a computation, while still being specific enough to be as helpful as possible to the user.

Herbgrind constructs symbolic expressions by first recording a *concrete* expression for every floating-point value and then using *anti-unification* to combine these concrete expressions into symbolic expressions. Each concrete expression tracks the floating-point operations that were used to build a particular value. Concrete expression are copied when a value is passed to a function, entered in a heap data structure, or stored and retrieved later, but these operations themselves are not recorded. A single concrete expression (and the resulting symbolic expression), might encompass a subtraction

[7]Herbgrind only reports those sources of error where error flows into spots, so that users are only shown erroneous code that affects program results.

$$
\begin{aligned}
&\mathrm{Addr} = \mathrm{PC} = \mathbb{Z} \\
&\mathcal{M}[a : \mathrm{Addr}] : \mathbb{F} \mid \mathbb{Z} \\
&\mathrm{pc} : \mathrm{PC} \\
&\mathrm{prog}[n : \mathrm{PC}] : \mathrm{Addr} \leftarrow f(\overrightarrow{\mathrm{Addr}}) \\
&\qquad\qquad \mid \textbf{if } P(\overrightarrow{\mathrm{Addr}}) \textbf{ goto } \mathrm{PC} \\
&\qquad\qquad \mid \textbf{out } \mathrm{Addr} \\
&\mathrm{run}(\mathrm{prog}) = \\
&\quad \mathcal{M}[-] = 0, \mathrm{pc} = 0 \\
&\quad \textbf{while } \mathrm{pc} \geq 0 \\
&\qquad \mathcal{M}, \mathrm{pc} = [\![\mathrm{prog}[\mathrm{pc}]]\!](\mathcal{M}, \mathrm{pc})
\end{aligned}
$$

$$
\begin{aligned}
&[\![y \leftarrow f(\vec{x})]\!](\mathcal{M}, \mathrm{pc}) = \\
&\quad r = [\![f]\!](\overrightarrow{\mathcal{M}[x]}) \\
&\quad \mathcal{M}[y \mapsto r], \mathrm{pc} + 1 \\
&[\![\textbf{if } P(\vec{x}) \textbf{ goto } n]\!](\mathcal{M}, \mathrm{pc}) = \\
&\quad \mathrm{pc}' = \textbf{if } [\![P]\!](\overrightarrow{\mathcal{M}[x]}) \textbf{ then } n \textbf{ else } \mathrm{pc} + 1 \\
&\quad \mathcal{M}, \mathrm{pc}' \\
&[\![\textbf{out } x]\!](\mathcal{M}, \mathrm{pc}) = \\
&\quad \mathrm{print}\ \mathcal{M}[x] \\
&\quad \mathcal{M}, \mathrm{pc} + 1
\end{aligned}
$$

Figure 2. The abstract machine semantics for low-level floating-point computation. A machine contains memory and a program counter which indexes into a list of statements. Statements either compute values and store the result in memory, perform a conditional jump, or output a value.

$$
\begin{aligned}
&\mathrm{Expr} = \mathbb{R} \mid f(\overrightarrow{\mathrm{Expr}}) \\
&\mathcal{M}_{\mathbb{R}}[a : \mathrm{Addr}] : \mathbb{R} \\
&\mathcal{M}_I[a : \mathrm{Addr}] : \mathrm{Set\ PC} \\
&\mathcal{M}_E[a : \mathrm{Addr}] : \mathrm{Expr} \\
&\mathrm{ops}[n : \mathrm{PC}] : \\
&\quad \mathrm{Set}\ (\mathrm{Expr} \times \mathrm{List}\,(\mathrm{Set}\,\mathbb{F}) \times (\mathrm{Position} \to \mathrm{Set}\,\mathbb{F})) \\
&\mathrm{spots}[n : \mathrm{PC}] : (\mathrm{Set}\,\mathbb{R}) \times (\mathrm{Set\ PC})
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{analyze}(\mathrm{prog}) = \\
&\quad \mathcal{M}[-] = \mathcal{M}_{\mathbb{R}}[-] = \mathcal{M}_E[-] = 0, \mathcal{M}_I[-] = \emptyset \\
&\quad \mathrm{pc} = 0, \mathrm{spots}[-] = (\emptyset, \emptyset), \mathrm{ops}[-] = \emptyset \\
&\quad \textbf{while } \mathrm{pc} \geq 0 \\
&\qquad \mathcal{M}', \mathrm{pc}' = [\![\mathrm{prog}[\mathrm{pc}]]\!](\mathcal{M}, \mathrm{pc}) \\
&\qquad \mathcal{M}'_{\mathbb{R}} = [\![\mathrm{prog}[\mathrm{pc}]]\!]_{\mathbb{R}}(\mathcal{M}, \mathcal{M}_{\mathbb{R}}) \\
&\qquad \mathcal{M}'_I = [\![\mathrm{prog}[\mathrm{pc}]]\!]_I(\mathcal{M}_{\mathbb{R}}, \mathcal{M}_I, \mathrm{pc}) \\
&\qquad \mathcal{M}'_E = [\![\mathrm{prog}[\mathrm{pc}]]\!]_E(\mathcal{M}'_{\mathbb{R}}, \mathcal{M}_E) \\
&\qquad \mathrm{record}(\mathrm{prog}, \mathrm{pc}, \mathrm{ops}, \mathrm{spots}, \mathcal{M}', \mathcal{M}_{\mathbb{R}}, \mathcal{M}'_I, \mathcal{M}'_E) \\
&\qquad \mathrm{pc} = \mathrm{pc}', \mathcal{M} = \mathcal{M}', \mathcal{M}_{\mathbb{R}} = \mathcal{M}'_{\mathbb{R}}, \mathcal{M}_I = \mathcal{M}'_I, \mathcal{M}_E = \mathcal{M}'_E \\
&\quad \textbf{return } \mathrm{spots}, \mathrm{ops}
\end{aligned}
$$

Figure 3. The Herbgrind analysis for finding root causes for floating-point error. Herbgrind maintains shadow memories for real values ($\mathcal{M}_{\mathbb{R}}$), influences ($\mathcal{M}_I$), and concrete expressions ($\mathcal{M}_E$). Additionally, Herbgrind tracks concrete expressions and input sets (both total and problematic) for operations in ops and error and influences for spots in spots. Note that Herbgrind follows floating-point control flow branches during analysis; cases when it diverges from the control flow interpreted under reals are reported as errors.

from one function, a series of multiplications from another, and an exponentiation which occurs after the value is stored in a hash table and later retrieved.

From the concrete expression for every value, Herbgrind computes symbolic expressions using a variant of the classic *anti-unification* algorithm [30] for computing the most-specific generalization of two trees. Symbolic expressions are much like concrete expressions, but include variables which can stand in for any subtree; variables which stand in for equivalent subtrees are the same. To produce expressions which are more useful for program improvement, Herbgrind uses a modified version of anti-unification. These modifications are described in an extended tech report. Reporting detailed symbolic expressions is essential for diagnosing the root causes of error; in the case of the plotter, the full extracted expression was essential for producing an improved complex square root definition.

4.4 Input Characteristics

Because floating-point error is non-uniform, the error of a computation is highly dependent on its inputs. In many cases, a developer must know on the range of inputs to a computation in order to improve its error behavior, but the actual intermediate inputs such computations receive during program execution are difficult to ascertain from the original, top-level program input. Herbgrind satisfies this need by computing *input characteristics* for each symbolic expression it produces.[8] These input characteristics can show the ranges of each symbolic variable, example inputs, or other features of the expression inputs.

To compute input characteristics, Herbgrind stores, for every symbolic expression, a summary of all values seen for that symbolic expression's free variables. Every time a section of code (a function or a loop body, say) is re-executed,

[8] Note that Herbgrind's input characteristics apply to the inputs of symbolic expressions identified by Herbgrind, not to the program inputs provided by the developer.

// *Reals*

$[\![y \leftarrow f(\vec{x})]\!]_{\mathbb{R}}(\mathcal{M}, \mathcal{M}_{\mathbb{R}})$ **when** $\mathcal{M}_{\mathbb{R}}[y] \in \mathbb{R}$ =

  $\vec{v}$ = **if** $\overrightarrow{\mathcal{M}_{\mathbb{R}}[x]} \in \mathbb{R}$ **then** $\overrightarrow{\mathcal{M}_{\mathbb{R}}[x]}$ **else** $\overrightarrow{\mathcal{M}[x]}$

  $\mathcal{M}_{\mathbb{R}}[y \mapsto [\![f]\!]_{\mathbb{R}}(\vec{v})]$

// *Influences*

$[\![y \leftarrow f(\vec{x})]\!]_{I}(\mathcal{M}_{\mathbb{R}}, \mathcal{M}_I, \text{pc})$ **when** $\mathcal{M}_{\mathbb{R}}[y] \in \mathbb{R}$ =

  $s = \bigcup \overrightarrow{\mathcal{M}_I[x]}$

  if local-error$(f, \overrightarrow{\mathcal{M}_{\mathbb{R}}[x]}) > T_\ell$ **then**

    $s = \text{pc} :: s$

  $\mathcal{M}_I[y \mapsto s]$

local-error$(f, \vec{v})$ =

  $r_{\mathbb{R}} = \mathbb{F}([\![f]\!]_{\mathbb{R}}(\vec{v}))$

  $r_{\mathbb{F}} = [\![f]\!]_{\mathbb{F}}(\overrightarrow{\mathbb{F}(v)})$

  $\mathcal{E}(r_{\mathbb{R}}, r_{\mathbb{F}})$

// *Expressions*

$[\![y \leftarrow f(\vec{x})]\!]_{E}(\mathcal{M}_{\mathbb{R}}, \mathcal{M}_E)$ **when** $\mathcal{M}_{\mathbb{R}}[y] \in \mathbb{R}$ =

  e = **if** $\overrightarrow{\mathcal{M}_{\mathbb{R}}[x]} \in \mathbb{R}$ **then** $f(\overrightarrow{\mathcal{M}_E[x]})$ **else** $\mathcal{M}_{\mathbb{R}}[y]$

  $\mathcal{M}_E[y \mapsto e]$

  update-problematic-inputs$(e, \hat{c})$

    nodes, positions =

      get-all-descendant-nodes(e)

    $\hat{c}'$ = make-table()

    for node, position **in** zip(nodes, positions) :

      $\hat{c}'$[position] = $\hat{c}$[position] + {node.value}

record(prog, pc, ops, spots, $\mathcal{M}, \mathcal{M}_{\mathbb{R}}, \mathcal{M}_I, \mathcal{M}_E$) =

  $e, \vec{c}, \hat{c}$ = ops[pc]

  ε, i = spots[pc]

  match prog[pc] **with**

  | $(y \leftarrow f(\vec{x}))$ **when** $\mathcal{M}[y] \in \mathbb{F} \implies$

    $e' = \mathcal{M}_E[y] :: e$

    $\vec{c}\,'$ = update-total-inputs$(\vec{x}, \vec{c})$

    if local-error$(f, \overrightarrow{\mathcal{M}_{\mathbb{R}}[x]}) > T_\ell$ **then**

      $\hat{c}'$ = update-problematic-inputs$(\mathcal{M}_E[y], \hat{c})$

    ops[pc] = $(e', \vec{c}\,', \hat{c}')$

  | $(y \leftarrow f(\vec{x}))$ **when** $\mathcal{M}[y] \in \mathbb{Z} \implies$

    if $[\![f]\!]_{\mathbb{R}}(\overrightarrow{\mathcal{M}_{\mathbb{R}}[x]}) = \mathcal{M}[y]$ **then**

      spots[pc] = $(1 :: \varepsilon, i \bigcup \overrightarrow{\mathcal{M}_I[x]})$

    else

      spots[pc] = $(0 :: \varepsilon, i)$

  | (**if** $P(\vec{x})$ **goto** y) $\implies$

    if $[\![P]\!]_{\mathbb{R}}(\overrightarrow{\mathcal{M}_{\mathbb{R}}[x]}) = [\![P]\!](\overrightarrow{\mathcal{M}[x]})$ **then**

      spots[pc] = $(1 :: \varepsilon, i \bigcup \overrightarrow{\mathcal{M}_I[x]})$

    else

      spots[pc] = $(0 :: \varepsilon, i)$

  | (**out** x) $\implies$

    $r = \mathcal{E}(\mathcal{M}[x], \mathcal{M}_{\mathbb{R}}[x])$

    if $r > T_m$ **then**

      spots[pc] = $(r :: \varepsilon, i \cup \mathcal{M}_I[x])$

    else

      spots[pc] = $(r :: \varepsilon, i)$

  update-total-inputs$(\vec{v}, \vec{c})$ =

    $\vec{c}\,'$ = []

    for v, c **in** zip$(\vec{v}, \vec{c})$:

      $\vec{c}\,' = (c + \{v\}) :: \vec{c}\,'$

    $\vec{c}\,'$ = reverse$(\vec{c}\,')$

Figure 4. On the left, the real-number execution, influence propagation, and concrete expressions building in Herbgrind; shadow executions not shown are no-ops. On the right, how Herbgrind updates the operation and spot information on every statement. Below are helper functions.

the inputs from that run are added to the summary. The input characteristics system is modular, and Herbgrind comes with three implementations.[9] In the first kind of input characteristic, a representative input is selected from the input. In the second kind of input characteristic, ranges are tracked for each variable in a symbolic expression. In the third kind of input characteristic, ranges are tracked separately for positive and negative values of each variable.

Herbgrind operates on a single execution of the client program using representative inputs provided by the developer. During execution problematic code fragments typically see a range of intermediate values, only some of which lead to output error.Herbgrind's input characteristics characterize that range of intermediate values, and thus rely on a well-chosen representative input for the program. For example, in our complex plotter example in the overview, the function `f` is run for every pixel in the image, fully exercising its input range. Since only some executions of a block of code lead to high local error, the input characteristics system provides two outputs for each characteristic and each expression: one for all inputs that the expression is called on, and one for all inputs that it has high error on.

The characteristics reported are tightly coupled to the symbolic expression for the relevant program fragment; each characteristic applies to a single variable in the expression. For instance, when the symbolic expression is `sqrt(x+1) - sqrt(x)`, the input characterization system might report that the variable `x` ranges from 1 to 1e20. This uses the specific variable name reported in the expression, and applies to both nodes labeled `x` in the expression. Since anti-unification

[9] The abstract Floatgrind analysis supports any arbitrary summary function on sets of input points, and for performance the summary function must be incremental.

guarantees that nodes assigned the same variable have taken the same values on each concrete expression, any valid summaries of the two nodes will be equivalent.

5 Implementation

The previous section described Herbgrind's analysis in terms of an abstract machine; however, important numerical software is actually written in low level languages like C, C++, and Fortran—sometimes a polyglot of all three.

To support all these use cases, we refine the algorithm presented in Section 4 to operate on compiled binaries instead of abstract machine programs. Herbgrind does this by building upon Valgrind [27], a framework for dynamic analysis through binary instrumentation. Building upon Valgrind requires mapping the abstract machine described in Section 4 to the VEX machine internal to Valgrind.

Analyses built upon Valgrind receive the instructions of the client program translated to VEX, and can add instrumentation to them freely before they are compiled back to native machine code and executed.

Implementing the algorithm from Section 4 with Valgrind requires adapting the abstract machine's notions of values, storage, and operations to those provided by Valgrind.

5.1 Values

Unlike the abstract machine, VEX has values of different sizes and semantics. Floating-point values come in different precisions, so the Herbgrind implementation makes a distinction between single- and double-precision floating-point values. Both types of values are shadowed with the same shadow state, but their behaviors in the client program are different, and they have different sizes in memory. Client programs also have file descriptors, pointers, and integers of various sizes, but this does not affect Herbgrind, since it does not analyze non-floating-point computations.

The algorithm in Section 4 tracks the exact value of computations by shadowing floating-point values with real numbers. Herbgrind approximates the real numbers using the MPFR library [13] to shadow floating-point values with arbitrary-precision floating-point.[10] As an alternative, we could use an efficient library for the computable reals [5, 23, 26].[11]

5.2 Storage

The abstract machine model of Section 4 represents storage as a single map from locations to values. However, VEX has three different types of storage—temporaries, thread state, and memory—and the latter two store unstructured bytes, not values directly. Herbgrind uses slightly different approaches for each.

[10]The precision used is configurable, set to 1000 by default.

[11]Herbgrind treats real computation as an abstract data type and alternate strategies could easily be substituted in.

To support SIMD instructions in the SSE instruction set, temporaries can contain multiple floating-point values, unlike the memory locations in the abstract machine. Herbgrind attaches a *shadow temporary* to each temporary: a type-tagged unit which can store multiple shadow values. The shadow values stored in a shadow temporary correspond to the individual floating-point values inside a SIMD vector. Temporaries that only store a single value have trivial shadow temporaries.

Thread state in VEX, which represents machine registers, is an unstructured array of bytes, so it does not use shadow temporaries. Each floating-point value consumes multiple bytes, and floating-point values of different sizes take up different numbers of bytes. This means that for reads and writes to memory, Herbgrind must be careful to check whether they overwrite nearby memory locations with shadow values. Herbgrind also supports writing SIMD results to memory and reading them back at an offset, as long as the boundaries of individual values are respected. In the rare cases where client programs make misaligned reads of floating-point values, Herbgrind conservatively acts as if the read computes a floating-point value from non-floating-point inputs.

Like thread state, memory is an unstructured array of bytes, with the complication that it is too large to shadow completely. Herbgrind shadows only memory that holds floating-point values; memory is shadowed by a hash table from memory addresses to shadow values.

5.3 Operations

In the abstract machine model, all floating-point operations are handled by specialized instructions. However, few machines support complex operations such as logarithms or tangents in hardware. Instead, client programs evaluate these functions by calling libraries, such as the standard libm. Shadowing these internal calculations directly would miscompute the exact value of library calls; Herbgrind therefore intercepts calls to common library functions before building concrete expressions and measuring error. For example, if a client program calls the tan function, Herbgrind will intercept this call and add tan to the program trace, not the actual instructions executed by calling tan.[12]

Expert-written numerical code often uses "compensating" terms to capture the error of a long chain of operations, and subtract that error from the final result. In the real numbers, this error term would always equal zero, since the reals don't have any error with respect to themselves. Yet in floating point, these "compensating" terms are non-zero and computations that produce them therefore have high local error. A naive implementation of Herbgrind would therefore report spots influenced by every compensated operation used to

[12]The library wrapping system in the implementation is extensible: users can add a new library call to be wrapped by appending a single line to a python source file.

compute it, even though the compensating terms increase the accuracy of the program.

Instead, Herbgrind attempts to detect compensating operations, and not propagate influence from the compensating term to the output of the compensated operation. Herbgrind identifies compensating operations by looking for additions and subtractions which meet two criteria: they return one of their arguments when computed in the reals; and the output has less error than the argument which is passed through. The influences for the other, compensating, term, are not propagated.

While most floating-point operations in real programs are specialized floating-point instructions or library calls, some programs use bitwise operations to implement a floating-point operation. Programs produced by gcc negate floating-point values by XORing the value with a bitmap that flips the sign bit, and a similar trick can be used for absolute values. Herbgrind detects and instruments these bitwise operations, treating them as the operations they implement (including in concrete expressions).

6 Optimization

A direct implementation of the algorithm in Section 5 is prohibitively expensive. Herbgrind improves on it by using the classic techniques of laziness, sharing, incrementalization, and approximation.

Laziness Since program memory is untyped, it is initially impossible to tell which bytes in the program correspond to floating-point values. Herbgrind therefore tracks floating-point values in memory lazily: as soon as the client program executes a floating-point operation on bytes loaded from a memory location, that location is treated as a floating-point location and shadowed by a new shadow value.

Besides lazily shadowing values in the client program, Herbgrind also minimizes instrumentation. Some thread state locations can always be ignored, such as CPU flag registers. VEX also adds a preamble to each basic block representing the control flow effects of the architecture, which Herbgrind also ignores.

For more fine-grained instrumentation minimization, Herbgrind makes use of static superblock type analysis. Values known to be integers do not have to be instrumented, and values known to be floating point can have type checking elided. This combination of static type analysis and dynamic error analysis is crucial for reducing Herbgrind's overhead. Unfortunately, Herbgrind must still instrument many simple memory operations, since values that are between storage locations but not operated on could have shadow values.

The static type analysis is also used to reduce reduce calls from the instrumentation into Herbgrind C functions. Valgrind allows the instrumentation to call into C functions provided by Herbgrind, which then compute shadow values, build concrete expressions, and track influences. However, calls from client program to host functions are slow. The static type analysis allows inlining these computations directly into VEX, avoiding a client-host context switch, because the type system tracks the size of values in thread state. Knowing the size means no type or size tests need to be done, so instrumentation can be inlined without requiring branches and thus crossing superblock boundaries. Inlining is also used for copies between temporaries, and for some memory accesses, where the inlined code must also update reference counts.

Sharing Many floating-point values are copies of each other, scattered in temporaries, thread state, and memory. Though copying floating-point values is cheap on most architectures, copying shadow values requires copying MPFR values, concrete expressions, and influence sets. To save time and memory, shadow values are shared between copies. Shadow values are reference counted to ensure that they can be discarded once they no longer shadow any floating-point values. The trace nodes stored in a shadow value are not freed along with the shadow value, since traces also share structure. Traces are therefore reference counted as well, with each shadow value holding a reference to its trace node, and each trace node holding references to its children.

Many shadow values are freed shortly after they are created. Related data structures, like trace nodes, are also allocated and freed rapidly, so memory allocation quickly becomes a bottleneck. Herbgrind uses custom stack-backed pool allocators to quickly allocate and free many objects of the same size.

Incrementalization The algorithm in Section 4 accumulates errors, concrete expressions, and operation inputs per-instruction, and summarizes all the results after the program finishes running. For long-running programs, this approach requires storing large numbers of ever-growing concrete expressions. The implementation of Herbgrind avoids this problem by aggregating errors (into average- and maximum-total and local errors) concrete expressions (into symbolic expressions) and inputs (into input characteristics) incrementally, as the analysis runs. This leads to both large memory savings and significant speed-ups.

This incrementalization does not change the analysis results since our implementation of anti-unification, used to aggregate concrete expressions, and summation, used to aggregate error, are associative.

6.1 Approximation

Herbgrind makes a sound, but potentially incomplete, approximation to the standard anti-unification algorithm to speed it up. Anti-unification requires knowing which pairs of nodes are equivalent, so that those nodes could be generalized to the same variable. Symbolic expressions can be trees hundreds of nodes deep, and this equivalence information must be recomputed at every node, so computing

these equivalence classes for large trees is a significant portion of Herbgrind's runtime. To limit the cost, Herbgrind exactly computes equivalence information to only a bounded depth for each node, 5 by default. In practice, this depth suffices to produce high-quality symbolic expressions. This depth bound also allows freeing more concrete program trace nodes, further reducing memory usage.

7 Case Studies

This section describes three examples where Herbgrind was used to identify the root causes of floating-point errors in numerical programs: in all three cases, the bugs were fixed in later versions of the software. Herbgrind's three major subsystems were crucial in detecting and understanding these bugs.

Gram-Schmidt Orthonormalization Gram-Schmidt orthonormalization transforms a collection of vectors into a orthogonal basis of unit vectors for their span. We investigated an implementation of Gram-Schmidt orthonormalization provided by the Polybench benchmark suite for numerical kernels. Polybench is provided in several languages; we used the C version of Polybench 3.2.1.

In the Gram-Schmidt orthonormalization kernel, Herbgrind detected a floating-point problem which it reported to have 64 bits of error, surprising for a vetted numerical benchmark. Upon investigating, we found that Gram-Schmidt decomposition is not well defined on the given inputs, resulting in a division by zero; Herbgrind reports the resulting `NaN` value as having maximal error. The fundamental problem is not in the Gram-Schmidt procedure itself but in its invocation on an invalid intermediate value. Luckily, Herbgrind provides, as its example problematic input to the computation in question, a zero vector, an invalid input to Gram-Schmidt orthonormalization. Herbgrind's input characteristics system was able to link the error in the program output to the root cause of an input violating the precondition of the orthnormalization procedure. Note that there was nothing wrong with the procedure itself, but rather its interaction with the program around it. Upon understanding the bug (and fixing it ourselves), we tested version 4.2.0 of Polybench and confirmed that this more recent version fixed the problem by changing the procedure that generated the vectors to ensure that valid inputs to Gram-Schmidt orthonormalization are produced.

PID Controller A proportional-integral-derivative controller is a control mechanism widely used in industrial control systems. The controller attempts to keep some *measure* at a fixed value. It runs in a loop, receiving the current value of the measure as input and outputting the rate at which to increase or decrease the measure. We investigated an adaptation of a simple PID controller which runs for a fixed number of iterations and with a fixed rate of change to the measure [9].

We initially ran Herbgrind on the PID controller expecting to find, perhaps, some floating-point error in the controller code itself. Instead, we found that Herbgrind was detecting a problem in the loop condition. To run the PID controller for a limited number of seconds, the program tests the condition `(t < N)`, where `N` is the number of seconds. The variable `t` is stored as a double-precision floating-point number, and is incremented by `0.2` on every iteration through the loop. As we experimented with different loop bounds, Herbgrind noticed that the condition, for some loop bounds, iterates once too many times. For example, if the loop bound is set to `10.0`, the loop executes 51 times, not 50 times, because adding `0.2` to itself 50 times produces a value $3.5 \cdot 10^{-15}$ less than 10. This bug is closely related to one that occurred in the Patriot missile defense system in 1992, resulting in the death of 28 people [36]. Herbgrind's automatic marking of all control flow operations as spots was necessary to detect the bug and link the inaccurate increment to its affect on control flow. Herbgrind was successfully able to trace back from error detected in the output of the program to the *root cause* of the error, the inaccurate increment; the output contained the source location of the erroneous compare and reported that it was influenced by the inaccurate increment. We notified the authors of the adapted PID controller and they confirmed the bug and identified a fix: incrementing the `t` variable by 1 instead of 0.2, and changing the test to `(t * 0.2 < N)`.

Gromacs Gromacs is a molecular dynamics package used for simulating proteins, lipids, and nucleic acids in drug discovery, biochemistry, and molecular biology. We investigated the version of Gromacs that ships with the SPEC CPU 2006 benchmark suite. Gromacs is a large program—42 762 lines of C, with the inner loops, which consume approximately 95% of the runtime, written in 21 824 lines of Fortran. We tested Gromacs on the test workload provided by CPU 2006, which simulates the protein Lysozyme in a water-ion solution.

During this run, Herbgrind reported an error in the routine that computes dihedral angles (the angle between two planes, measured in a third, mutual orthogonal plane). For inputs where the dihedral angle is close to flat, corresponding to four colinear molecules, the dihedral angle computation was returning values with significant error due to cancellation in the computation of a determinant. These cases, though a small subset of all possible angles, were important. First, collections of four colinear molecules are common, for example in triple-bonded organic compounds such as alkynes. Second, molecular dynamics is chaotic, so even small errors can quickly cause dramatically different behavior.

Herbgrind's symbolic expression system was crucial in understanding the root cause of this bug. The dihedral angle procedure invokes code from multiple source files, across both C and Fortran, moving data into and out of vector data

structures. The symbolic expression gathered together the slivers of computation that contributed to the high rounding error. From the expression reported by Herbgrind the potential for cancellation was clear and the input characteristics provided by Herbgrind allowed us to narrow our investigations to flat angles. We identified the problem and developed a fix based on the numerical analysis literature [33]. After we contacted the developers, they confirmed the bug and explained that they had deployed a similar fix in recent GROMACS versions.

8 Evaluation

This section shows that Herbgrind identifies correct root causes of error in inaccurate floating-point program binaries, and that the root causes are reported with sufficient precision to allow improving accuracy. The first subsection demonstrates this for Herbgrind in its default configuration, while the second subsection examines the effect of Herbgrind's various tunable parameters. Each experiment uses the standard FPBench suite of general-purpose floating-point programs [10].

8.1 Improvability

The true root cause of a floating-point inaccuracy is a part of the inaccurate computation which can be rewritten to reduce error. Herbgrind's value is its ability to find true root causes; thus, this evaluation measures the fraction of true root causes found by Herbgrind, and the fraction of Herbgrind's candidate root causes that are true root causes.

Methodology To determine whether a candidate is a true root cause, one must determine whether its error can be improved, which depends on the expertise of the programmer. As a mechanical, easily quantifiable proxy, we picked a state-of-the-art tool, Herbie [29], and used it to determine which of Herbgrind's identified root causes were improvable. To use Herbgrind on the FPBench benchmarks, these benchmarks must be compiled to native code. We do so by using the publicly available FPCore-to-C compiler provided with FPBench, and then compiling this C code, along with some driver code which exercises the benchmarks on many inputs, using the GNU C Compiler. We then run the binary under Herbgrind, and pass the resulting output to Herbie. We also timed the benchmarks to measure their speed.[13]

All experiments were run on an Intel Core i7-4790K processor at 4GHz with 8 cores, running Debian 9 with 32 Gigabytes of memory.[14]. Herbgrind introduces a 574x overhead on the FPBench suite.

Not every benchmark in the FPBench suite exhibits significant error, nor can Herbie improve all of the benchmarks that exhibit error. To provide a valid comparison for Herbgrind's results, we compare Herbgrind against an "oracle" which directly extracts the relevant symbolic expression from source benchmark.

Results The oracle finds that, of 86 benchmarks, 30 have significant error (> 5 bits). Of these, 29 are determined by Herbgrind to have significant error.

Of the 30 benchmarks with significant error, the oracle produces an improvable root cause for all 30 benchmarks.

Herbgrind determines candidate root causes for 29 of the errors (96%), and for 25 of the benchmarks, Herbie detects significant error in the candidate root causes reported (86%). The remaining programs reflects either limitations in Herbgrind's ability to properly identify candidate root causes[15], or limitations in Herbie's ability to sample inputs effectively. Finally, of the 30 total benchmarks which had error detectable by the oracle and Herbie, Herbgrind can produce improvable root causes for 25 (83%).

Overall, Herbgrind is able to determine the true root cause for 25 of the programs in the FPBench suite, demonstrating that it is useful for diagnosing and fixing floating-point inaccuracies.

8.2 Subsystems

Herbgrind's analysis consists of three main subsystems, influence tracking, symbolic expressions, and input characteristics. In this section we will demonstrate the effect of these subsystems.

Figure 5a shows the result of running Herbgrind with various error thresholds for the influences system. The error threshold selected determines how much local error an expression has to have before it is marked as "significantly erroneous", and tracked as a source of error.

A higher threshold means that fewer computations are reported as being problematic. Users might want to use a higher error threshold on certain applications when there are too many expressions that are somewhat erroneous to address them all. In highly critical applications, where it is important to address even a small source of error, users might choose to lower the error threshold to catch even more errors.

Because of the non-local nature of floating-point error, the root cause of error is often spread across many floating-point operations. To measure how far root causes are spread, we varied the maximum expression depth Herbgrind tracked. In Figure 5c and Figure 5d, we measured the runtime and effectiveness of Herbgrind using various maximum expression depths.

A maximum expression depth of 1 node deep effectively disables symbolic expression tracking, and only reports the

[13] Our original timing code actually had a floating point bug, which we discovered when Herbgrind included it in its output.

[14] Results were obtained using GNU Parallel [35]

[15] Candidate root causes in which Herbie can not independently detect error mostly reflect limitations in Herbgrind's ability to accurately characterize inputs. Note that the input characterization system is modular and easily extended.

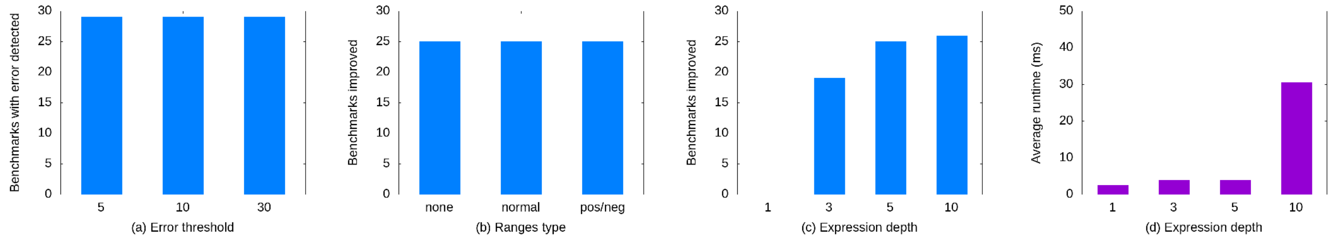

Figure 5. In (a) we compare the number of computations flagged with various error thresholds. In (b) we show how many benchmarks can be improved with various types of ranges. (c) and (d) explore the effect of different maximum expression depths on runtime and number of benchmarks improved.

operation where error is detected, much like FpDebug and similar floating-point debugging tools. However, unlike those tools, it still tracks the influence of error and the range of inputs to the operation. As you can see from the figure, not tracking operations before the one that produced error results in a speedup over the normal configuration, but at a high cost: none of the expressions produced are significantly improvable.

Finally, to test the effectiveness of input characterization, we measured the improvability of our benchmarks in three configurations: with ranges turned off , with a single range for all inputs, and with separate ranges for positive and negative inputs (see Figure 5b). In this dataset it appears that input ranges do not significantly affect results; however, this could be due to the fact that these programs are small micro-benchmarks.

Library Wrapping Herbgrind instruments calls to mathematical library functions such as sqrt and tan to correctly evaluate the exact result of a computation and provide simpler symbolic expressions. With this wrapping behavior turned off, Herbgrind finds significantly more complex expressions, representing the internals of library functions: the largest expressions are not 9 but 31 operations in size, and 133 expressions[16] have more than 9 operations. For example, instead of $e^x - 1$, Herbgrind finds 17 expressions such as

$$(x-0.6931472(y-6.755399\text{e}15)+2.576980\text{e}10)-2.576980\text{e}10.$$

Furthermore, as discussed in Section 5, without wrapping calls to mathematical libraries, Herbgrind measures output accuracy incorrectly, though on the FPBench examples the inaccuracy is slight.

8.3 Handling Expert Tricks

Detecting compensating terms (see Section 5.3) in client code is important for reducing false positives in Herbgrind's output. To test the compensation detection system, we applied Herbgrind to analyze Triangle, an expert-written numerical program.

Triangle [32], written in C by Jonathan Shewchuk, is a mesh generation tool, which computes the Delaunay triangulation of a set of input points, and can add additional points so that the triangulation produced satisfies various stability properties, such as avoiding particularly sharp angles. Running on Triangle's example inputs, we found that Herbgrind's compensation detection correctly handles all but 14 of 225 compensating terms with local error and does not present these false positives to the user.

The 14 remaining compensated operations are not detected, because the compensating term affects control flow: Triangle checks whether the compensating term is too large, and if so runs the same computation in a different way. Herbgrind's real-number execution computes the accurate value of a compensating term to be 0, so these branches often go the "wrong way". Fortunately, given Herbgrind's candidate root cause, this behavior is always easy to check in the Triangle source.

9 Related Work

There is a rich literature on analyzing and mitigating floating-point error. Below we discuss the most closely related work.

Recently, work on statically analyzing error for floating point programs has made tremendous progress [6, 8, 11, 14, 15, 24, 34]. Broadly, this work focuses on providing sound, though conservative, error bounds for small numerical kernels. This work is useful for reasoning about the expressions returned by Herbgrind, but does not on its own scale to large numerical software.

Several papers in recent years have used dynamic analyses to analyze floating-point error.

FpDebug [4] uses Valgrind to build a dynamic analysis of floating point error. Like Herbgrind, it uses MPFR shadow values to measure the error of individual computations. Unlike FpDebug however, Herbgrind's shadow real execution is based on a model of full programs, including control flow, conversions, and I/O, as opposed to FpDebug's model of VEX blocks. This enables a rigorous treatment of branches as spots, and leads to extensions such as wrapping library

[16] With library wrapping disabled, Herbgrind identifies 848 problematic expressions, mostly corresponding to false positives in the internals of the math library.

functions, sharing shadow values, SIMD operations, and bit-level transformations. All these features required significant design and engineering to scale to 300 KLOC numerical benchmarks from actual scientific computing applications. In addition to an improved real execution, Herbgrind departs from FpDebug with its spots and influences system, symbolic expressions, and input ranges, which allow it to connect inaccurate floating-point expressions to inaccurate outputs produced by the program. Herbgrind's use of local error, symbolic expressions, and input ranges, help the user diagnose the parts of the program that contributed to the detected error.

Similarly to FpDebug and Herbgrind, Verrou [12] is a dynamic floating point analysis built on Valgrind. Verrou's aim is also to detect floating point error in numerical software, but attempts to do so at much lower overhead. The resulting approach uses very little instrumentation to perturb the rounding of a floating point program, thus producing a much more conservative report of possible rounding errors.

Recent work by Bao and Zhang [3] also attempts to detect floating-point error with low overhead, with the goal of determining when floating-point error flows into what they call "discrete factors". The tool is designed to detect the possible presence of inaccuracy with very low runtime overhead to enable re-running in higher precision. In this context, a very high false positive rate (> 80-90% in their paper) is acceptable, but it is not generally acceptable as a debugging technique. Bao and Zhang's discrete factors address only floating-point errors that cause changes in integer or boolean values (hence "discrete"). Unlike [3], Herbgrind tracks all factors (not just discrete ones), including changes in floating-point values that lead to changes in floating-point outputs. Re-running programs in higher precision is untenable in many contexts, but may work for some.

Herbie [29] is a tool for the automatically improving the accuracy of small floating point expressions (≈ 10 LOC). Herbie uses randomly sampled input points and an MPFR-based ground truth to evaluate expression error. This statistical, dynamic approach to error cannot give sound guarantees, but is useful for guiding a search process. Herbie's main focus is on suggesting more-accurate floating-point expressions to program developers. Herbie can be combined with Herbgrind to improve problematic floating point code in large numerical programs, by feeding the expressions produced by Herbgrind directly into Herbie to improve them.

Wang, Zou, He, Xiong, Zhang, and Huang [37] develop a heuristic to determine which instructions in core mathematical libraries have an implicit dependence on the precision of the floating-point numbers. A ground truth for such precision-specific operations cannot be found by evaluating the operations at higher precision. These results justify Herbgrind detecting and abstracting calls to the floating-point math library.

CGRS [7] uses evolutionary search to find inputs that cause high floating-point error; these inputs can be used for debugging or verification. Unlike Herbgrind, these inputs can be unrealistic for the program domain, and CGRS does not help the developer determine which program expressions created the high error. However, users who want to analyze the behavior of their programs on such inputs can use Herbgrind to do so.

10 Conclusion

Floating point plays a critical role in applications supporting science, engineering, medicine, and finance. This paper presented Herbgrind, the first approach to identifying candidate root causes of floating point errors in such software. Herbgrind does this with three major subsystems: a shadow taint analysis which tracks the influence of error on important program locations, a shadow symbolic execution which records the computations that produced each value, and an input characterization system which reports the inputs to problematic computations. Herbgrind's analysis is implemented on top of the Valgrind framework, and finds bugs in standard numerical benchmarks and large numerical software written by experts.

References

[1] Micah Altman, Jeff Gill, and Michael P. McDonald. 2003. *Numerical Issues in Statistical Computing for the Social Scientist.* Springer-Verlag. 1–11 pages.

[2] Micah Altman and Michael P. McDonald. 2003. Replication with attention to numerical accuracy. *Political Analysis* 11, 3 (2003), 302–307. http://pan.oxfordjournals.org/content/11/3/302.abstract

[3] Tao Bao and Xiangyu Zhang. 2013. On-the-fly Detection of Instability Problems in Floating-point Program Execution. *SIGPLAN Not.* 48, 10 (Oct. 2013), 817–832. https://doi.org/10.1145/2544173.2509526

[4] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems *(PLDI '12)*. ACM, New York, NY, USA, 453–462. http://doi.acm.org/10.1145/2254064.2254118

[5] Hans-J. Boehm. 2004. The constructive reals as a Java Library. *J. Log. Algebr. Program* 64 (2004), 3–11.

[6] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846

[7] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamarić, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-point Errors. ACM, 43–52.

[8] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. 2015. Formal Methods for Industrial Critical Systems: 20th International Workshop, FMICS 2015 Oslo, Norway, June 22-23, 2015 Proceedings. (2015), 31–46.

[9] N. Damouche, M. Martel, and A. Chapoutot. 2015. Transformation of a {PID} Controller for Numerical Accuracy. *Electronic Notes in Theoretical Computer Science* 317 (2015), 47 – 54. https://doi.org/10.1016/j.entcs.2015.10.006 The Seventh and Eighth International Workshops on Numerical Software Verification (NSV).

[10] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. 2016. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. (July 2016).

[11] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals *(POPL '14)*. ACM, New York, NY, USA, 235–248. http://doi.acm.org/10.1145/2535838.2535874

[12] François Févotte and Bruno Lathuilière. 2016. VERROU: Assessing Floating-Point Accuracy Without Recompiling. (Oct. 2016). https://hal.archives-ouvertes.fr/hal-01383417 working paper or preprint.

[13] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2 (June 2007), 13:1–13:15. http://doi.acm.org/10.1145/1236463.1236468

[14] Eric Goubault and Sylvie Putot. 2011. Static Analysis of Finite Precision Computations *(VMCAI'11)*. Springer-Verlag, Berlin, Heidelberg, 232–247. http://dl.acm.org/citation.cfm?id=1946284.1946301

[15] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[16] Andreas Jaeger. 2016. OpenLibm. http://openlibm.org/

[17] W. Kahan. 1965. Pracniques: Further Remarks on Reducing Truncation Errors. *Commun. ACM* 8, 1 (Jan. 1965), 40–. https://doi.org/10.1145/363707.363723

[18] William Kahan. 1971. A Survey of Error Analysis.. In *IFIP Congress (2)*. 1214–1239. http://dblp.uni-trier.de/db/conf/ifip/ifip71-2.html#Kahan71

[19] W. Kahan. 1987. Branch Cuts for Complex Elementary Functions or Much Ado About Nothing's Sign Bit. In *The State of the Art in Numerical Analysis (Birmingham, 1986)*, A. Iserles and M. J. D. Powell (Eds.). Inst. Math. Appl. Conf. Ser. New Ser., Vol. 9. Oxford Univ. Press, New York, 165âĂŞ211.

[20] W. Kahan. 1998. *The Improbability of Probabilistic Error Analyses for Numerical Computations*. Technical Report. 34 pages. http://www.cs.berkeley.edu/~wkahan/improber.pdf

[21] William Kahan. 2005. Floating-Point Arithmetic Besieged by "Business Decisions". World-Wide Web lecture notes.. , 28 pages. http://www.cs.berkeley.edu/~wkahan/ARITH_17.pdf

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.* 5, 3 (Sept. 1979), 308–323. https://doi.org/10.1145/355841.355847

[23] Vernon A. Lee, Jr. and Hans-J. Boehm. 1990. Optimizing Programs over the Constructive Reals. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 102–111. https://doi.org/10.1145/93542.93558

[24] Matthieu Martel. 2009. Program Transformation for Numerical Precision *(PEPM '09)*. ACM, New York, NY, USA, 101–110. http://doi.acm.org/10.1145/1480945.1480960

[25] B. D. McCullough and H. D. Vinod. 1999. The Numerical Reliability of Econometric Software. *Journal of Economic Literature* 37, 2 (1999), 633–665.

[26] Marvin L. Minsky. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[27] Nethercote and Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. (June 2007).

[28] Dr. K-C Ng. 1993. FDLIBM. http://www.netlib.org/fdlibm/readme

[29] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM.

[30] Gordon D. Plotkin. 1970. A note on inductive generalization. *Machine Intelligence* 5 (1970), 153–163.

[31] Kevin Quinn. 1983. Ever Had Problems Rounding Off Figures? This Stock Exchange Has. *The Wall Street Journal* (November 8, 1983), 37.

[32] Jonathan Richard Shewchuk. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, Ming C. Lin and Dinesh Manocha (Eds.). Lecture Notes in Computer Science, Vol. 1148. Springer-Verlag, 203–222. From the First ACM Workshop on Applied Computational Geometry.

[33] Hang Si. 2015. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Trans. Math. Softw.* 41, 2, Article 11 (Feb. 2015), 36 pages. https://doi.org/10.1145/2629697

[34] Alexey Solovyev, Charlie Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions *(FM'15)*. Springer.

[35] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. http://www.gnu.org/s/parallel

[36] U.S. General Accounting Office. 1992. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. http://www.gao.gov/products/IMTEC-92-26

[37] Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2015. Detecting and Fixing Precision-Specific Operations for Measuring Floating-Point Errors *(FSE'15)*.

[38] Debora Weber-Wulff. 1992. Rounding error changes Parliament makeup. http://catless.ncl.ac.uk/Risks/13.37.html#subj4