A Dynamic Program Analysis to find Floating-Point Accuracy Problems

Florian Benz
Saarland University
fbenz@stud.uni-saarland.de

Andreas Hildebrandt
Johannes-Gutenberg Universität Mainz
andreas.hildebrandt@uni-mainz.de

Sebastian Hack
Saarland University
hack@cs.uni-saarland.de

Abstract

Programs using floating-point arithmetic are prone to accuracy problems caused by rounding and catastrophic cancellation. These phenomena provoke bugs that are notoriously hard to track down: the program does not necessarily crash and the results are not necessarily obviously wrong, but often subtly inaccurate. Further use of these values can lead to catastrophic errors.

In this paper, we present a dynamic program analysis that supports the programmer in finding accuracy problems. Our analysis uses binary translation to perform every floating-point computation side by side in higher precision. Furthermore, we use a lightweight slicing approach to track the evolution of errors.

We evaluate our analysis by demonstrating that it catches well-known floating-point accuracy problems and by analyzing the Spec CFP2006 floating-point benchmark. In the latter, we show how our tool tracks down a catastrophic cancellation that causes a complete loss of accuracy leading to a meaningless program result. Finally, we apply our program to a complex, real-world bioinformatics application in which our program detected a serious cancellation. Correcting the instability led not only to improved quality of the result, but also to an improvement of the program's run time.

Categories and Subject Descriptors G.1.0 [*Numerical Analysis*]: General—Computer arithmetic, Error analysis, Multiple precision arithmetic

Keywords Dynamic program analysis, program instrumentation, floating-point accuracy

1. Introduction

Floating-point numbers are almost always mere approximations of the "true" real numbers: there is an uncountably infinite number of real values in any open interval on the real line, but any digital computer provides only finite storage for their representation. Due to this approximation, floating-point arithmetic is prone to accuracy problems caused by insufficient precision, rounding errors, and catastrophic cancellation. In fact, writing a *numerically stable* program is challenging. However, few programmers are aware of the intricacies that come with floating-point arithmetic. And even if a programmer is familiar with the problem per se, he will often be unable to detect or even prevent it: theoretical results often allow only a worst-case analysis that soon becomes overly pessimistic for complex cases. We lack the proper tools to support the programmers in efficiently and effectively finding floating-point problems at runtime. In the past, floating-point accuracy problems caused program failures that cost millions of dollars or even human life. And even if a numerical inaccuracy does not result in such catastrophic events, it can cause serious harm if it remains undetected: in many application scenarios, correct results can hardly be distinguished from incorrect ones. In such cases, an undetected numerical inaccuracy in a popular application can lead to countless flawed results.

One of the best-studied examples of an undetected accumulation of rounding errors was the failure of the MIM-104 Patriot air defense system in the Gulf War. As a result 28 US Army soldiers were killed by an Iraqi Scud missile on February 25, 1991. A government investigation [18] revealed that the system's internal clock had drifted by 0.3 seconds after operating for 100 hours. This was due to a rounding error when converting integer values to floating-point values.

Another well-studied example occurred at the Vancouver Stock Exchange. As reported by the Wall Street Journal [20], the index of the Stock Exchange was initialized at 1000 points in January 1982.

For each trade, the index was updated by first adding the weighted stock price to the previous index and then truncating the result to three decimal places. This operation occurred about 3000 times a day and the accumulated truncations led to a loss of around one point per day. The error remained undiscovered for 22 months. After the discovery, the index was corrected from 524.811 to 1098.892 points. The calculation of the index was then changed from truncating to rounding.

Both examples drastically demonstrate that floating-point issues can have dramatic consequences. However, there is only a surprisingly little number of tools that assist the programmers (who are usually totally unaware of the intricacies of floating-point arithmetic) in tracking down these problems *although* the need for easily usable tools for the dynamic analysis of numeric stability has been recognized in the literature and cast in the form of demands (e.g. by Kahan [12]). In the past, both static and dynamic program analyses have been proposed. Static analyses like Fluctuat [5] provide sound over-approximations of floating-point problems such as rounding errors. However, they suffer from imprecision and are heavily dependent on other analyses (e.g. pointer analyses) to disambiguate the data flow of a program. Therefore, static techniques work exceptionally well in special domains like embedded systems where the code does usually not exhibit complex and dynamic data structures. For large, possibly object-oriented programs, however, they are less suitable. In contrast, the dynamic program analyses proposed in the past [1, 2, 13] do either not scale to large-scale systems or make compromises in their analysis power to achieve scalability.

In this paper, we bridge this gap by proposing a dynamic program analysis that detects strictly more problems than previous

PLDI'12, June 11–16, 2012, Beijing, China.

approaches and scales to relevant real-world applications. In summary, we make the following contributions:

- Our analysis simulates real arithmetic by performing each floating-point computation in higher precision side by side to the original calculation. For every floating-point variable in the program, we introduce a *shadow variable* in higher precision. We use those shadow values to detect catastrophic cancellations and rounding errors. In contrast to a static analysis, we do not detect all such errors but, because of the higher precision of the shadow values, we detect significantly more than all previous dynamic analyses. For the sake of precision, we formally describe our dynamic program analysis by means of an operational semantics.
- In addition to the shadow value, we collect additional data for every floating-point operation. This information is used by a light-weight slicing technique which helps the programmer to reconstruct the path of operations along which an error propagated after the program run. This helps in localizing the source of inaccuracies that propagated to other program points where they were actually detected.
- We implemented the presented analysis in the Valgrind [16] binary instrumentation framework. We evaluate the effectiveness of our tool on pathological floating-point accuracy problems from the literature as well as on large-scale real-world programs: the SPEC CFP2006 benchmark suite of floating-point programs, the Biochemical Algorithms Library (BALL), and the GNU linear programming kit (GLPK). In all investigated programs, our program detects accuracy problems with various consequences, ranging from unnecessarily slow convergence to meaningless output. We are not aware of any previous *automated* study of floating-point accuracy problems on programs of that scale. Finally, we also present experiments that show the limits of our approach.

The rest of this paper is structured as follows: The next section summarizes the foundations of floating-point arithmetic relevant for this paper and gives an overview of several important classes of accuracy problems. Section 3 outlines the concepts of the analysis, Section 4 presents how the programmer interacts with the analysis, and Section 5 discusses issues of a concrete implementation of the analysis. Finally, in Section 6 we present an extensive case study of our analysis.

2. Foundations

We briefly repeat the necessary foundations of IEEE 754 floating-point arithmetic in this section. We follow the notation and terminology of Goldberg [8] and Higham [10]. The interested reader is referred to these publications for more detail.

2.1 The IEEE 754 Standard

A floating-point number has the form

$$x = \pm s \times \beta^e$$

where for each value, the sign, the significant s (also called mantissa), and the exponent e are stored. IEEE 754-1985, the arguably most important floating point standard, uses $\beta = 2$. The *precision* is the maximum number of digits which can be represented with the significant. IEEE 754-1985 defines four precisions: single, single-extended, double, and double-extended. Here we only consider single and double precision since these are the most common in today's programming languages.

As the representation uses only a finite number of bits, an error due to the approximation is unavoidable. Denoting the evaluation of an expression in floating-point arithmetic by the function fl, rounding can be represented by the transformation $x \mapsto fl(x)$. Rounding to the nearest representable number is the default and the only rounding mode considered here. In case of a tie, it is rounded to the nearest even number. The unit roundoff u, also called the machine epsilon, is defined as the largest positive number such that

$$fl(1+u) = 1$$

The unit roundoff depends on the base β and the precision p:

$$u = \beta/2 \cdot \beta^{-p}$$

Let $\mathbb{F} \subset \mathbb{R}$ be the set of numbers which can be exactly represented by floating-point arithmetic. If $x \in \mathbb{R}$ lies in the range of $\mathbb{F}$, then

$$fl(x) \quad = \quad x(1+\varepsilon) \quad |\varepsilon| < u$$

For all $a, b \in \mathbb{F}$, the following properties hold

$$\begin{array}{llll} fl(a \circ b) & = & (a \circ b)(1+\varepsilon) & |\varepsilon| \leqslant u \quad \circ \in \{+,-,\times,/\} \\ fl(a \circ b) & = & fl(a \circ b) & \circ \in \{+,\times\} \end{array}$$

The standard requires that all basic operations are computed as if done exactly and rounded afterwards. This is stronger than the properties above as it requires $\varepsilon = 0$ if $a \circ b \in \mathbb{F}$. In floating-point arithmetic, addition and multiplication are commutative but no operation is associative.

2.2 Error Sources

The errors sources can be separated into three groups: rounding, data uncertainty, and truncation.

Rounding errors are unavoidable due to the finite precision. *Uncertainty* in the data comes from the initial input or the result of a previous computation. Input data from measurements or estimations is usually only accurate to a few digits. *Truncation* arises when a numerical algorithm approximates a mathematical function. Many numerical methods take finitely many terms of a Taylor-, Laurent-, or Fourier series; the terms omitted constitute the truncation error. A truncation error can only be analyzed with knowledge about the function the algorithm computes. Therefore, a tool working on the program as is can only track rounding errors and uncertainty from previous computations.

Usually, the discrepancy between a floating-point number and the corresponding exact number is measured as a relative error. Throughout this paper, the following definition is used

$$\text{relative error} = \left| \frac{\text{exact value} - \text{approximate value}}{\text{exact value}} \right|$$

Note that if the exact value is 0, the relative error is ∞.

2.2.1 Error Accumulation

A good example for a number which cannot be exactly represented in floating-point arithmetic is the decimal number 0.1, which is periodical in the binary system

$$(0.1)_{10} = \left(0.000\overline{1100}\right)_2$$

The relative error of 0.1 in single precision is close to the unit roundoff of 2^{-24} ($\approx 5.96 \times 10^{-8}$)

$$\left| \frac{(0.1)_{10} - (0.1)_{sp}}{(0.1)_{10}} \right| \approx 5.94 \times 10^{-8}$$

The error introduced by an approximate representation of a constant is in the same range as the rounding error after a computation. This error cannot be avoided because a representation of a real number with a finite number of bits can never be exact for all numbers.

Figure 1 shows a C program with large error accumulation. In the end, the value of `time` is 1999.6588. Thus, `time` has a

```
float time = 0.0f;
int i;
for (i = 0; i < 20000; i++) {
  time += 0.1f;
}
```

Figure 1. C program with large error accumulation

relative error of 1.7×10^{-4}. This is significantly higher than the error of the constant `0.1f`. If the same program is run with `time` in double precision and the constant is still represented in single precision (i.e. only replacing `float` with `double`) the relative error of `time` decreases to 1.5×10^{-8}. If, in addition to the double precision variable `time`, the constant is also represented in double precision (i.e. removing the `f` behind the constant) the relative error is reduced to 3.6×10^{-13}.

In general, an error introduced by a constant limits the accuracy of the result. The actual influence depends on the precision and the numerical stability of the algorithm. An algorithm is *numerically stable* if the influence of rounding errors to the result is provably always small. Thus, numerical stability is a property of an algorithm. A similar property, called the condition, describes the problem the algorithm tries to solve. The condition is independent from the algorithm and describes the dependency between the input and the output of a problem. The *condition number* is a measurement for the asymptotically worst case of how strongly changes in the input can influence the output. If small changes in the input lead to large changes in the output, the condition number is large. Thus, it is easier to design algorithms for problems with low condition numbers. In the example above, it does not matter if the error comes from a constant, because always adding the same number with an error tends to accumulate the error. Large errors usually occur due to the insidious growth of just a few rounding errors and it is hard to find their origins.

2.2.2 Insufficient Precision

The precision used in a program is usually a trade-off between performance and sufficient accuracy. The example program in Figure 2 uses a constant which is slightly smaller than the unit roundoff.

```
/* float unit roundoff is 0.00000006f */
float e = 0.00000005f;
float sum = 1.0f;
int i;
for (i = 0; i < 5; i++) {
  sum += e;
}
```

Figure 2. C program with an error due to insufficient precision

In the end, the value of `sum` is 1.0. Thus, the relative error of `sum` is 2.5×10^{-7}. Any precision slightly higher than single precision leads to a correct result. Insufficient precision is usually hard to discover manually because the error introduced by a single operation is small. However, large errors can occur due to error accumulation. Often, double precision is thus used to alleviate these problems. However, while double precision indeed suffices for many application scenarios, it can still be insufficient, depending on the details of the problem.

2.2.3 Catastrophic Cancellation

If two nearby quantities are subtracted, the most significant digits match and cancel each other. This effect is called cancellation and can be *catastrophic* or *benign*. A cancellation is catastrophic if at least one operand is subject to rounding. Otherwise, if both quantities are exactly known, the cancellation is benign. A cancellation can occur when subtracting or adding two numbers because both operands can be positive or negative. An example for a cancellation is the following computation

$$1.002 \times 10^3 - 1.000 \times 10^3 = 2.000 \times 10^0$$

This cancellation is catastrophic if the digit 2 in the first operand is the result of an error and benign otherwise. An operation with a cancellation can expose an error introduced by an earlier computation, but the same operation can be exact if no error has been introduced before. Some formulas can be rearranged to replace a catastrophic cancellation with a benign one, but only within the scope of the particular formula, i.e., this cannot guard against rounding errors present in the input. One example is the expression $x^2 - y^2$, where x^2 and y^2 are likely to be subject to rounding. In this case, a more accurate formula would be $(x - y)(x + y)$, because here, the subtraction is done directly on the variables x and y. Assuming that x and y are exactly known, a possible cancellation is thus guaranteed to be benign in the second variant.

The number of canceled digits can be calculated with the following formula

$$\max\{\text{exponent}(op_1), \text{exponent}(op_2)\} - \text{exponent}(res)$$

where op_1 and op_2 are the operands, and res the result. A cancellation has happened if the number of canceled digits is greater than zero. To conclude, for the analysis of floating-point software, it is crucial to decide whether a cancellation is catastrophic or benign.

2.3 Cancellation of Rounding Errors

Inaccurate intermediate results can still lead to an accurate final result. Therefore, not all inaccurate values indicate a problem.

One example from Higham [10] for a calculation where more inaccurate intermediate results lead to a more accurate final result is the calculation of $(e^x - 1)/x$ with a unit roundoff $u = 2^{-24}$. To obtain a more accurate result, it is better to compute $(e^x - 1)/\ln(e^x)$. For $x = 9 \times 10^{-8}$ the exact result is 1.00000005. The following calculations show that the second formula has more inaccurate intermediate results but the final result has a relative error that is 1.92×10^7 times lower than the error of the first formula

$$\begin{aligned} fl\left(\tfrac{e^x-1}{x}\right) &= fl\left(\tfrac{1.19209290\times10^{-7}}{9.00000000\times10^{-8}}\right) = 1.32454766 \\ fl\left(\tfrac{e^x-1}{\ln(e^x)}\right) &= fl\left(\tfrac{1.19109290\times10^{-7}}{1.19209282\times10^{-7}}\right) = 1.00000006 \end{aligned}$$

2.4 Summary

The dynamic program analysis we present in this paper addresses all issues discussed in this section:

- Error accumulation and insufficient precision is detected by our analysis by following values through the whole program with a side-by-side computation in higher precision. In the end, the accumulation and insufficient precision can be detected by comparing the original and the shadow value. Of course, our analysis is not correct in the sense that we pretend that the shadow values in higher precision are perfect substitutes for the exact real values. However, our experience shows that side-by-side calculation in higher precision helps in detecting severe accuracy problems.
- Our analysis detects catastrophic cancellations by calculating the canceled bits and the bits that are still exact for every floating-point addition and subtraction. This is explained in Section 3.1.
- The cancellation of rounding errors can lead to false positives in intermediate results. Our analysis addresses this problem by determining variables that are likely the final result and thus

helps the user to determine if the final result is affected. Error traces created by the analysis can then reveal that the inaccurate intermediate results led to an accurate final result. The error trace is explained in Section 3.2.

3. The Analysis

In the following, we describe a dynamic program analysis which assists the programmer in finding accuracy problems. The analysis' main ingredient is a side-by-side computation done with an arbitrary but fixed precision[1] which is higher than the precision used in the client program. The analysis tracks two kinds of information: For every *original value* that is computed by the program (this comprises variables which are written to memory as well as temporary results kept in registers), it stores a *shadow value* which consists of several components among which is a counterpart of the original value in higher precision. Additionally, we compute analysis information for every executed floating-point instruction. Our analysis essentially tracks the difference of a floating-point value in the client program and its corresponding shadow value. If this difference becomes too large, it is likely that the client program suffers from accuracy problems. The exact composition of the analysis information is shown in the first section of Figure 6.

We use two different sets of information (per variable, per instruction) because they are both relevant: Accuracy problems usually materialize as inaccurate contents of variables. From there, it is helpful to find the instructions that cause the problem and potentially reconstruct how the error propagated to that variable.

An important detail is that the side-by-side computation in our analysis does not affect the semantics of the program: For example, a comparison which decides whether a branch is taken can have a different outcome in higher precision than in the original precision. We would detect the accuracy problem but still follow the branch based on the result of the computation in original precision. As a consequence, the result of the analysis is *not* equal to the result of the program as if it would have been executed in higher precision. This is the desired behavior, because we want to track down problems in the program running in the original precision.

The results of the side-by-side computation are leveraged in two ways (further detailed in the following sections): By treating the shadow values as a potentially better approximation to the exact real value, we compute the relative error and the cancellation badness of every instruction. The latter detects catastrophic cancellations *and* indicates how much more precise a value would have to be for the cancellation to be benign (see Section 3.1). Furthermore, we use the shadow values to calculate the relative error on every instruction and original value. This information can be used to find the cause of the accuracy problem (see Section 3.2).

Finally, we want to stress that a dynamic program analysis is never complete. The capability of the presented analysis to detect floating-point accuracy problems is based on the assumption that the shadow values are a substitute for the real exact values.

3.1 Cancellation Detection

Because cancellations occur often and benign cancellations have no impact on the accuracy, it is important to detect if a cancellation is catastrophic or benign. To this end, we define *cancellation badness* as an indicator for catastrophic cancellations. The cancellation badness relates the number of bits canceled by an operation to the bits that where exact *before* the operation. The exact bits of a floating-point variable v are determined with respect to the "exact value" $\tilde{v}$ which is in our analysis stored in the shadow value of v as a floating-point number in higher precision[2]:

$$\mathrm{ebits}(\tilde{v}, v) := \begin{cases} p & \text{if } \tilde{v} = v \\ 0 & \text{if } \mathrm{exponent}(\tilde{v}) \neq \mathrm{exponent}(v) \\ \mathrm{ebits}'(\tilde{v}, v) & \text{otherwise} \end{cases}$$

with

$$\mathrm{ebits}'(\tilde{v}, v) := \min\{p, |\mathrm{exponent}(\tilde{v}) - \mathrm{exponent}(\tilde{v} - v)|\}$$

Consider an instruction $v' \leftarrow v_1 \diamondsuit v_2$. The number of bits canceled by executing the instruction is calculated by

$$cbits := \max\{\mathrm{exponent}(v_1), \mathrm{exponent}(v_2)\} - \mathrm{exponent}(v')$$

The cancellation badness $cbad$ now relates the least number of exact bits of the operands to the number of canceled bits:

$$cbad := \max\{0, 1 + cbits - \min\{\mathrm{ebits}(\tilde{v}_1, v_1), \mathrm{ebits}(\tilde{v}_2, v_2)\}\}$$

If the badness is greater than zero, a catastrophic cancellation happened. The value of $cbad$ itself indicates how much more precise the operands would have to be for the cancellation to not be catastrophic. If the badness is 0, the cancellation was benign.

Consider the following example:

$$1.379 - 1.375 = 0.004$$

Here, three digits are canceled. Assuming that the first operand has four exact digits and the second operand has three exact digits, the cancellation badness is one. Therefore, the cancellation is catastrophic and the result is completely inaccurate. If the same calculation is done with a second operand with four exact digits, the cancellation badness is zero and thus the cancellation is benign. Assuming that the exact fourth digit is 8, the accurate result

$$1.379 - 1.378 = 0.001$$

has only the exponent in common with the inaccurate result above.

For every instruction the sum of the cancellation badness of every execution of that instruction (field scb in an instruction's analysis information in Figure 6) and the maximum cancellation badness (mcb) that occurred over all executions of that instruction are maintained. Each shadow value tracks the maximum cancellation badness that occurred in the corresponding original value (field mcb in an shadow value's analysis information in Figure 6) and a pointer to the instruction where this maximum occurred (mcb_src).

3.2 Finding Operations that Cause Errors

Consider the pathological example in Figure 3. Naïvely, we would

```
1  float e = 0.00000006f;
2  float x = 0.5f;
3  float y = 1.0f + x;
4  float more = y + e;
5  float diff_e = more - y;
6  float diff_0 = diff_e - e;
7  float zero = diff_0 + diff_0;
8  float result = 2 * zero;
```

Figure 3. A program with insufficient precision and catastrophic cancellation.

expect `result` to be `0.0` after executing line 8. However, the operations in line 5 and 6 cause catastrophic cancellations which render the computed values inexact. This error propagates into variable `result` which is not `0.0` at the end. However, the operation on which `result` depends, the addition in line 7, does not contribute to this inaccuracy. All exact bits of `diff_0` are preserved. Hence, the programmer here wants to find the operations in line 5 and 6. Of

[1] The precision can be specified by the user before the analysis starts.

[2] In the following, the tilde stands for operations and numbers (and sets thereof) in higher precision.

course, in large programs, the operations contributing to the problem can be spread over several modules. Thus, it is hard to pinpoint the problem by just looking at the source code.

A dynamic slice (a data dependence graph of the execution) of the execution would help the programmer to locate this problem. However, recording the trace of the whole program execution and computing the relative errors afterwards can drastically slow down the analysis, because the result of every operation has to be written to disk. Instead, we pursue a more light-weight approach: For every instruction in the program, we maintain two values: The sum of the relative error of every execution (sre in Figure 6) of that instruction and the maximum relative error (mre) that occurred over all executions of that instruction.

Please note that for a shadow value of zero, the concept of a relative deviation becomes meaningless. To guard against such cases, our implementation allows the user to specify a threshold t_0 such that

$$\text{relative error} = \frac{|\text{shadow value} - \text{original value}|}{\max\left(|\text{shadow value}|, |t_0|\right)}$$

The sum over all relative errors is used to compute an average of the relative errors. If an instruction has a small average relative error, it is usually unlikely that this instruction is involved in significant accuracy problems. Hence, this value is used when the analysis results are presented to the user: Instructions with a comparably high average relative error are listed before instructions with a lower average relative error.

The maximum relative error is used to enable a light-weight slicing approach: Whenever the maximum relative error of an instructions changes, i.e. the instruction is executed again with a relative error that is higher than all the relative errors seen in previous executions, we store the instructions which computed the arguments to that instruction (field mre_src in an instruction's analysis information in Figure 6). In this way, we obtain insight into the origins of the values causing the maximum error. After the analysis run, this information is used to reconstruct a slice along which the maximum error propagated.

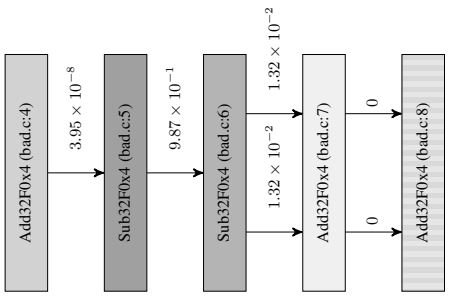

Figure 4. Error trace for the variable `result` in the truncated C program on the left which suffers from insufficient precision in line 4 and catastrophic cancellations in lines 5 and 6. The edges are labeled with the introduced error of the operation they flow out of.

To reveal the origins of an error, one has to find the operations which are the cause. Therefore, after the analysis run, the contribution of an operation to the maximum relative error is calculated. The contribution is called the *introduced error* of an operation. The introduced error is calculated as the smallest difference of the maximum error of the operation itself and the maximum error of one of the operations which is a direct predecessor in the error trace. Special cases occur when an operation is its own predecessor or when an operation has no predecessor. In these cases, the maximum error of the operation itself is taken instead of the difference. Note that the introduced error can be negative, because errors can be canceled. Figure 4 shows an error trace for the example above. The edges in the graph are labeled with the introduced error of the operation producing the value that flows on the edge. One can directly see that the subtraction in line 5 introduces a large error of 9.87×10^{-1}.

Note that the error trace is not necessarily a part of the dynamic data-dependence graph of the program run. This is because the maximum relative errors of the instructions in the error trace might be observed at *different* instances of the instructions. Hence, the data flow suggested in the error trace might not have taken place in the execution of the program, since we only update the source of the error when the maximum relative error changes (see Figure 6). However, in our experiments we made the experience that the error trace very often resembles the actual data flow and is thus very helpful for getting a first impression on how an error propagates through a program.

3.3 Operational Semantics

In this section, we formally describe our analysis using a structural operational semantics. We base our formalization on Valgrind's VEX intermediate representation language, mainly because the tool implementing our analysis is implemented in Valgrind. However, the subset of VEX we use here is so generic that it does not affect the applicability of our analysis in other frameworks. Figure 5 shows the excerpt of the VEX grammar relevant for our analysis.

const	::=	constant
temp	::=	temporary variable
exp	::=	temp | const
stmt	::=	temp := load(exp) | store(exp, exp)
		| temp := get(exp) | put(exp, exp)
		| temp := exp | tmp := $\diamondsuit_u$ exp
		| temp := exp $\diamondsuit_b$ exp
program	::=	stmt*

Figure 5. Simplified grammar of the VEX intermediate representation used in Valgrind. Load and store describe memory accesses whereas put and get are for accessing registers.

The VEX intermediate representation consists of a sequence of numbered statements. Valgrind flattens the intermediate representation to simplify the instrumentation. As a result, an expression is only a temporary variable or a constant and can not be a tree of expressions. Statements consist of memory and register writes, and assignments. The right side of an assignment can be a memory read, register read, temporary variable, constant, unary operation, or binary operation.

Figure 6 summarizes the formalization of our analysis. Its first section shows the analysis information for instructions and variables. Elements of analysis information are stored in various maps: μ_t (μ_m) implements the store that maps temporaries (memory addresses) to original values. Δ_t (Δ_m) implements the store that maps temporaries (memory addresses) to shadow values. Ω maps addresses of instructions to elements of instruction analysis information. The notation $A[x \leftarrow y]$ stands for $\lambda w.w = x \ ? \ y : A(w)$.

In the inference rule section, we only show the rules for load, store, and the binary operation. Each inference rule is of the form:

$$\frac{\begin{array}{c}\text{side-by-side computation}\\ \text{original computation}\end{array}}{\langle\text{configuration}\rangle\ \text{instruction} \rightsquigarrow \langle\text{configuration}'\rangle\ \text{instruction}'}$$

The rules for put and get are purely technical and do mostly resemble the ones for load and store. Most interesting is the rule for the binary operator and the computation of the various analysis information components outlined in the last section of the figure. For

Analysis information tuple I for an instruction:

Name	Domain	Description
sre	$\widetilde{\mathbb{F}}$	Sum of relative errors
mre	$\widetilde{\mathbb{F}}$	Maximum relative error
mre_src	$\mathbb{N} \times \mathbb{N}$	Pair of instruction addresses that point to the instructions that computed the operands that caused the maximum relative error
scb	$\mathbb{N}$	Sum of cancellation badnesses
mcb	$\mathbb{N}$	Maximum cancellation badness

Shadow value tuple S for a variable:

Name	Domain	Description
val	$\widetilde{\mathbb{F}}$	Value in higher precision
td	$\mathbb{N}$	Depth of the computation tree
pc	$\mathbb{N}$	Address of the instruction that wrote to the variable lastly
mcb	$\mathbb{N}$	Maximum cancellation badness
mcb_src	$\mathbb{N}$	Instruction that caused maximum cancellation badness

$$\begin{array}{rcll} \Sigma & : & Addr \to Instr & \text{Maps an address to an instruction} \\ \mu_m & : & Addr \to \mathbb{F} & \text{Maps an address to an original value} \\ \mu_t & : & Temp \to \mathbb{F} & \text{Maps a temporary to an original value} \\ \Omega & : & Addr \to I & \text{Maps an address to an instruction analysis information} \\ \Delta_m & : & Addr \to S & \text{Maps an address to a shadow value} \\ \Delta_t & : & Temp \to S & \text{Maps a temporary to a shadow value} \end{array}$$

$$\frac{s = \Delta_m[a] \quad \Delta_t{}' = \Delta_t[n' \leftarrow s] \quad a = \mu_t[n] \quad v = \mu_m[a] \quad \mu_t{}' = \mu_t[n' \leftarrow v] \quad \iota = \Sigma[pc+1]}{\langle \mu_t, \mu_m, \Delta_t, \Delta_m, \Sigma, pc\rangle\, n' := \mathrm{load}\,(n) \rightsquigarrow \langle \mu_t{}', \mu_m, \Delta_t{}', \Delta_m, \Sigma, pc+1\rangle\, \iota}\ \text{F-LOAD}$$

$$\frac{s = \Delta_t[n_2] \quad \Delta_m{}' = \Delta_m[a \leftarrow s] \quad a = \mu_t[n_1] \quad v = \mu_t[n_2] \quad \mu_m{}' = \mu_m[a \leftarrow v] \quad \iota = \Sigma[pc+1]}{\langle \mu_t, \mu_m, \Delta_t, \Delta_m, \Sigma, pc\rangle\, \mathrm{store}\,(n_1, n_2) \rightsquigarrow \langle \mu_t, \mu_m{}', \Delta_t, \Delta_m{}', \Sigma, pc+1\rangle\, \iota}\ \text{F-STORE}$$

$$\frac{\Delta_t{}' = \Delta_t[n' \leftarrow s'] \quad \Omega' = \Omega[pc \leftarrow o'] \quad v_1 = \mu_t[n_1] \quad v_2 = \mu_t[n_2] \quad v' = v_1 \diamondsuit v_2 \quad \mu_t{}' = \mu_t[n' \leftarrow v'] \quad \iota = \Sigma[pc+1]}{\langle \mu_t, \mu_m, \Delta_t, \Delta_m, \Sigma, pc\rangle\, n' := n_1 \diamondsuit n_2 \rightsquigarrow \langle \mu_t{}', \mu_m, \Delta_t{}', \Delta_m, \Sigma, pc+1\rangle\, \iota}\ \text{F-BINOP}$$

Abbreviations used:

$$\begin{array}{rcl} \text{rel. error } rerr & := & |(s'.val - v')\,/d| \\ d & := & s'.val \neq 0\ ?\ s'.val : t_0 \\ \\ \mathrm{lookup}(n, v) & := & \exists w.\,(n, w) \in \Delta_t\ ?\ \Delta_t[n] : init \\ init & := & (v, 0, -, 0, -) \end{array}$$

Analysis information update for an instruction:

$$\begin{array}{lcl} o & = & \Omega[pc] \\ o'.sre & = & o.sre + rerr \\ o'.mre & = & \max\{o.mre, rerr\} \\ o'.mre\_src & = & rerr > o.mre\ ?\ (s_1.pc, s_2.pc) \\ & & :\ o.mre\_src \\ o'.scb & = & o.scb + cbad \\ o'.mcb & = & \max\{o.mcb, cbad\} \end{array}$$

Shadow value of the result of a binary operator:

$$\begin{array}{lcl} s_1 & = & \mathrm{lookup}(n_1, v_1) \\ s_2 & = & \mathrm{lookup}(n_2, v_2) \\ s'.pc & = & pc \\ s'.val & = & s_1.val \mathbin{\widetilde{\diamondsuit}} s_2.val \\ s'.td & = & 1 + \max\{s_1.td, s_2.td\} \\ s'.mcb & = & \max\{s.mcb, cbad\} \\ s'.mcb\_src & = & cbad > s.mcb\ ?\ pc : s.mcb\_src \end{array}$$

Figure 6. Structural operational semantics for the original and the side-by-side computation

the sake of brevity, we also omit the unary operator. Conceptually, it equals the binary operator. However, computations that involve both operands of a binary operator (such as the calculation of the maximum cancellation badness) have to be adapted accordingly.

4. User Interface

Our tool can automatically collect information for the whole execution of a client program. Thereby all floating-point instructions and variables are analyzed and variables are determined that are likely the final result by the depth of the execution tree leading to them (field td in an shadow value's analysis information in Figure 6).

In addition, our implementation enables the user to perform a more targeted analysis by adding client requests to the source code of the analyzed program; client request are instructions that are only interpreted by our analysis and ignored in an execution without Valgrind.

With the client request, the user can exclude parts of the program from the analysis or can run the analysis only on specified parts. This not only allows to specify which parts are to be analyzed, but also grants access to the information gathered by the analysis during execution. Thus, via client requests, the relative error of a variable can be accessed and the error trace of a variable can be constructed at every point in the program where the variable is present.

Furthermore, an original value can be replaced with its shadow value, so that the user can correct a value without fixing all the code leading to this value in the client program. Thus, the user is given a simple way to check whether the correction of a value influences the result of a program.

Another powerful instrument are stages (originally proposed by Kahan [12]). The basic idea behind this concept is to add break points, called "stages", into a program, to observe the errors at each break point, and to complain if two consecutive stages differ too much. This is a semi-automatic technique that assumes that the programmer has an idea where interesting stages are located. In practice, interesting stages almost always correspond to loop iterations. Using our tool, the programmer can define multiple sets of stages that are analyzed independently. An example where stages prove to be useful is presented in Section 6.1. This example shows that stages help finding accuracy problems even if the shadow values on their own fail to do so because of their finite precision.

5. Implementation Details

The analysis presented in Section 3 instruments every floating-point machine instruction in the program. However, our early experiments showed that this is not sufficient. The basic reason is that on the assembly level, strict typing is no longer present. As a result, the types used by Valgrind do not necessarily match the type used in the source code of the client program. An example is shown in Figure 7. There, the compiler chose to load the bit pattern representing a floating-point constant into an integer register whose contents is then stored to memory. Later on, that memory cell is read into an SSE register and a floating-point operation is performed on the loaded value.

This means that the type with which constants are stored in memory does not have to match the type with which they are loaded. A naïve implementation would not only instrument the floating-point but also all integer instructions. However, this would result in a slow and error-prone implementation with a large memory overhead.

Instead, we start the instrumentation as late as possible. As a result, a floating-point variable is tracked from the point where it

```
movl $0x3356bf95, %eax      movl $0x3356bf95, %eax
movl %eax, -4(%rbp)         movl %eax, -4(%rbp)
...                         ...
addss -4(%rbp), %xmm0       fadds -4(%rbp)
```

Figure 7. Extracts from the assembly code for a SSE addition (left) and a x87 FPU addition (right) of two floating-point numbers. There are no indications that the loaded values are used in floating-point instructions later on.

is the result of a floating-point operation. For a constant or a value given as an input to the program there is no need for a shadow value as the shadow value would be exactly the same. In the operational semantics in Figure 6 this is expressed by the function lookup. Furthermore, we use copy propagation to avoid instrumentation of instructions that do not affect shadow values such as type cast operations. However, all reads and writes to the main memory have to be instrumented as in each such operation a floating-point value could be involved.

This scheme has the advantage that it does not only simplify instrumentation and increases the efficiency of the analysis but it also makes the analysis more robust. Floating-point operations which are not yet instrumented can only lead to undetected errors but do not affect the correctness of our analysis. This is due to the fallback to the original value if no shadow value exits.

6. Evaluation

The capabilities and the performance of the tool are evaluated in this section. Unless stated otherwise, all analyses are performed on an x86-64 system with a shadow value precision of 120 bit.

6.1 Wrong Limit of Convergent Sequence

The sequence u_n converges to 6, but if computed with any finite precision it converges to 100. The sequence is defined as

$$u_n = \begin{cases} 2 & \text{if } n = 0 \\ -4 & \text{if } n = 1 \\ 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}} & \text{if } n > 1 \end{cases}$$

This sequence has been analyzed by Kahan [12] and Muller et al. [15]. The strange behavior can be explained with the solution of the recurrence

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^{n} + \beta \cdot 6^{n} + \gamma \cdot 5^{n}}$$

where α, β, and γ depend on u_0 and u_1. For $u_0 = 2$ and $u_1 = -4$ one gets $\alpha = 0$, $\beta = -3$, and $\gamma = 4$. However, if computed with finite precision, roundoff errors cannot be avoided and result in an α that is not exactly zero. This leads to the different limits.

```
double u = 2, v = -4, w;
for (int i = 3; i <= 100; i++)
{
  FPDEBUG_BEGIN_STAGE(0);

  w = 111. - 1130./v + 3000./(v*u);
  u = v;
  v = w;
  printf("u%d = %1.17g\n", i, v);

  FPDEBUG_END_STAGE(0);
}
FPDEBUG_PRINT_ERROR(&"u_100", &v);
```

Figure 8. C code which tries to compute the sequence u_n

Therefore, it is clear that after analyzing the computation of u_n in Figure 8, no error can be observed because the shadow value is also 100. However, both values converge at a different rate which can be observed with stages. The stage report in Figure 9 reveals that the relative error grows faster than linear in the iterations 4, 18 and in all iterations in between. Thus, the stage concept allows the detection of this strange behavior which would otherwise have gone unnoticed.

```
Stage 0:

(0)   0x7ff000200 (15)
      executions: [4, 18]
      origin: 0x40089A
```

Figure 9. The stage report for the code in Listing 8 reveals problems starting in iteration 4 and ending in iteration 18. As 15 problems are reported, all iterations between 4 and 18 are affected.

6.2 Walker's Floating-Point Benchmarks

Walker [19] provides two floating-point benchmarks called fbench and ffbench.

The program *fbench* is an implementation of a complete optical design ray tracing algorithm. The largest relative error discovered is 9×10^{-13}. Thus, the algorithm used in fbench does not suffer from accuracy problems for the sample data. The program comes with its own version of the trigonometric functions used, but there is no difference in stability if the program is run with the trigonometric functions from the C library.

The second benchmark, *ffbench*, performs a two-dimensional Fast Fourier transform. Only 4 out of 131,089 shadow values have an error. The largest relative error is 9.6×10^{-2}. Two additions produce inaccuracies due to catastrophic cancellation. However, the inaccuracy stays local and has no influence on the final result.

A program where more than 99% of the variables are accurate is rare. Even considering that ffbench computes and then inverts a Fast Fourier transform, so that the program should result in the same value as the initial one, it requires that no significant digits are lost in between. Therefore, it depends on the numerical stability of the algorithm and on the initial values.

6.3 Analysis of expf

Let us now give an example of the limitations of our approach: The function `expf` from the GNU C Library computes e^x in single precision. The implementation of the function analyzed here is from `sysdeps/ieee754/flt-32/e_expf.c` (GNU C Library 2.12.2). Executing the function `expf` in higher precision leads to more inaccurate results because the algorithm used is designed for single precision and uses precomputed values.

The function is based on Gal's accurate tables method [6]. First, n, t, and y are computed such that x can be expressed as

$$x = n \cdot \ln(2) + \frac{t}{512} + \mathrm{delta}\,[t] + y$$

and then e^x can be approximated as

$$e^x \approx 2^n e^{\frac{t}{512} + \mathrm{delta}[t]} \cdot (1 + \mathrm{p}\,(y + \mathrm{delta}\,[t] + n \cdot \ln(2)))$$

where p is a second-degree polynomial approximating $e^x - 1$ and $\mathrm{delta}\,[t]$ and $e^{\frac{t}{512} + \mathrm{delta}[t]}$ are obtained from tables.

The problem arises when computing n and t. The value of n is computed with

$$n = x \cdot \frac{1}{\ln(2)} + \mathrm{THREEp22} - \mathrm{THREEp22}$$

where $\mathrm{THREEp22} = 12582912.0$ is a single precision constant and the operations are performed with single precision.

The remaining part dx of x is computed with

$$dx = x - n \cdot \ln(2)$$

One would expect $n \cdot \ln(2)$ to be equal to x. But for $x = 0.09$ the value of n is 0 because the large value of THREEp22 leaves no significant digits of x when added. This results in $dx = x = 0.09$ but the shadow value is 1.27×10^{-9}. The shadow value is not exactly 0 because $\frac{1}{\ln(2)} \cdot \ln(2)$ is not exactly 1 as the first factor is a single precision constant and the second factor a double precision one. The value of t is computed with

$$\frac{t}{512} = dx + \text{THREEp42} - \text{THREEp42}$$

where THREEp42 = 13194139533312.0 is a single precision constant but the operations are performed with double precision.

With the previous value of dx, one gets $\frac{t}{512} \approx x$ for the original value but $\frac{t}{512} = 0$ for the shadow value. Thus, t converted to an integer is 46 in the original program but 0 in the side-by-side computation.

Because of these differences the values taken from the precomputed table do not fit for the shadow value. For $x = 0.09$ this results in a relative error between the original and the shadow value of 1.56×10^{-4}. However, the original value only has a relative error of 2.18×10^{-8} compared to the exact result. This relative error is smaller than the single precision unit roundoff and thus cannot be more accurate.

6.4 SPECfp2006

SPECfp2006 is the floating-point component of the SPEC CPU2006 benchmark suite [3]. Most of the floating-point benchmarks from SPECfp2006 can be analyzed with our tool. The only exceptions are `zeusmp` which fails because Valgrind does not support data segments bigger than 1 GB, and `dealII` which does not terminate because it relies on the internal 80-bit representation of the x87 FPU. Valgrind however behaves IEEE 754 compliant and only works with 64 bits for double precision even if it is an x87 FPU instruction. The reasons for the problems were discovered by Weaver [21].

SPECfp2006 comes with three data sets for each benchmark: train, test, and ref. The "ref" data sets are the largest. Because of the slowdown caused by the instrumentation overhead, we focus on the "test" dataset.

All benchmarks were performed on a quad-core AMD Opteron processor with 2.5 GHz and 64 GB of RAM. Figure 10 shows the results for all SPECfp2006 benchmarks we executed. The side-by-side computation ran with a precision of 120 bit.

Benchmark	Original	Analyzed	Slowdown
`bwaves`	47.5 s	7920 s	167 x
`gamess`	0.7 s	381 s	544 x
`milc`	30.9 s	7860 s	224 x
`gromacs`	2.1 s	991 s	472 x
`cactusADM`	4.7 s	4777 s	1016 x
`leslie3d`	59.8 s	17467 s	292 x
`namd`	19.8 s	18952 s	957 x
`soplex`	0.027 s	5.0 s	185 x
`povray`	0.9 s	400.0 s	444 x
`calculix`	0.07 s	17.1 s	244 x
`GemsFDTD`	5.5 s	1146 s	208 x
`tonto`	1.26 s	404 s	321 x
`lbm`	9.55 s	2893 s	303 x
`wrf`	7.68 s	2623 s	342 x
`sphinx3`	4.41 s	938 s	213 x

Figure 10. Results of the SPECfp2006 benchmarks with the "test" data sets

6.4.1 Analysis of CalculiX

During our benchmarking, we observed a potential loss of accuracy in the benchmark CalculiX. Figure 11 shows the first entry of the mean error file which is sorted by the introduced error. This entry led to a deeper analysis of DVdot, a function in SPOOLES, a linear equation solver used by CalculiX, to compute the dot product of two vectors. As it turns out, it is not the multiplication which causes large inaccuracies, but rather the addition in this line. This shows that a globally computed introduced error is not a perfect indicator for the real origin of the problem. Computed only with knowledge about the function DVdot, the introduced errors reveal the real origin.

```
DVdot (Utilities_DV.c:245) Mul64F0x2 (116,010)
  avg error:  1.70635239241881 * 10^1
  max error:  1.55742240304371 * 10^6
  cancellation badness - max: 0, avg: 0.00
  introduced error (max path):  1.55.. * 10^6
```

Figure 11. Information about the operation with the largest introduced error in CalculiX. The output contains information about the function name, the place in the source code, the VEX IR operation name, the execution count, the average and the maximum relative error and average and maximum cancellation badness of all executions of the operation in comparison to the side-by-side computation, and the introduced error computed with the preceding operations.

After spotting a potential problem, we manually modified the function `DVdot` in SPOOLES to print out input causing large inaccuracies and to see if the insertion of a more accurate value influences the final result (see in Figure 12). The reset causes the deletion of all shadow values and thus, no previous inaccuracies influence the observation. If one starts the tool with the analysis disabled, then the begin and end client requests cause that only DVdot is observed and one obtains a good output of the mean errors of the operations in DVdot. After the computation of `sum`, the error of the variable is checked against the error bound of 10^{-2}. If the error is greater, the error and the content of the arrays are printed. At the end of the function, the variable `sum` is set to the shadow value. As the shadow value provides a significantly better result, this is a correction of the value of `sum`.

```
double DVdot(int size, double y[], double x[]) {
  FPDEBUG_RESET();
  FPDEBUG_BEGIN();

  double sum = 0.0;
  for (int i = 0; i < size; i++) {
    sum += y[i] * x[i];
  }

  double errorBound = 1e-2;
  if (FPDEBUG_ERROR_GREATER(&sum, &errorBound)) {
    FPDEBUG_PRINT_ERROR(&"sum", &sum);
    for (int j = 0; j < size; j++) {
      printf("x[%d] = %.20g\n", j, x[j]);
      printf("y[%d] = %.20g\n", j, y[j]);
    }
  }
  FPDEBUG_INSERT_SHADOW(&sum);
  FPDEBUG_END();

  return sum;
}
```

Figure 12. Shortened DVdot function with error detection and correction

During a run with the provided "test" data set, the function `DVdot` produces a relative error greater than 10^{-2} in 28 cases. In

three of these cases the relative error is even greater than 10^{-1}. This means that in the worst case, no digit is correct. In that case, the original and the shadow value differ by a factor of nearly 1.76.

The error correction influences 8.5% of the floating-point numbers in the output of CalculiX. Some of the numbers differ in every digit. The "test" data set describes a cantilever beam under shear forces. The correction does not affect the computed stresses but the computed displacements. However, all of the influenced displacements are smaller than 10^{-10}; therefore, the influence may be negligible for this simulation. Nevertheless, one sees that the naïve implementation of an inconspicuous mathematical function can have an impact on the accuracy of a whole computation.

The whole mean error file, sorted by the introduced error, is shown in Figure 13. The multiplication only has operands which are seen as "exact" and thus only produces a relative error smaller than the double precision unit roundoff of 2^{-53} ($\approx 1.11 \times 10^{-16}$). Whereas the addition receives the results of the multiplications as an input and produces large errors due to catastrophic cancellation.

```
DVdot (Utilities_DV.c:248) Add64F0x2 (116,010)
  avg error:  2.72694149800805 * 10^-4
  max error:  1.61405640329709 * 10^0
  cancellation badness - max: 3, avg: 0.01
  introduced error (max path):  1.61.. * 10^0

DVdot (Utilities_DV.c:248) Mul64F0x2 (116,010)
  avg error:  3.96985521613801 * 10^-17
  max error:  1.10661002489201 * 10^-16
  cancellation badness - max: 0, avg: 0.00
  introduced error (max path):  1.10.. * 10^-16
```

Figure 13. Mean errors of floating-point operations in DVdot

The average relative error of the addition is large because a difference between the original and the shadow value can be introduced by a single addition but is considered by every addition later on.

6.5 BALL

Our original motivation for developing a dynamic analysis framework for numerical stability came from the field of structural bioinformatics, where huge amounts of often noisy data are routinely analyzed using highly complex numerical methods. In this section, we present our first application of the developed tool to a subset of the Biochemical Algorithms Library (BALL) [11], a rapid application development framework for structural bioinformatics.

Our first experiments were performed on version 1.4 of BALL and were focused on the optimization of molecular structures against the Amber force field. To this end, we used the program `add_hydrogens` which is distributed along with BALL's source code. First, `add_hydrogens` reads a molecular input structure from a PDB-file, performs a number of required pre-processing steps (such as normalization of atom names and types, inference of missing bonds and of missing atoms). It then sets up one of currently three force fields (we used the default, Amber96) and minimizes the potential energy with respect to the atomic coordinates of the newly added atoms (in a typical use-case, these will mostly be hydrogens) using one of a number of configurable optimizers (we kept the default conjugate gradient for 500 optimization steps).

A global analysis of `add_hydrogens` reports several potential problematic areas in BALL's functionality for structure optimization, i.e., in the Amber implementation and the molecular optimizer. These areas have been analyzed in-depth, but only one area has an accuracy problem that verifiably influences the final result. However, this area includes calls to methods which are reported as potentially problematic but are not problematic if analyzed alone.

The main problem encountered is a catastrophic cancellation in a double precision subtraction. In the worst case, 23 bits are canceled. While this number would be relatively unproblematic at double precision with 53 bit, the operands here are both results of single precision computations and thus can have at most 24 exact bits. Therefore, it is not surprising that the cancellation badness is greater than zero.

The inaccuracy affects the computation of the conjugate gradient computation, i.e., the search for a descent direction, in computing the minimum energy conformation of a molecular system. If the inaccuracy is fixed by computing the numbers used in the subtraction in double precision, the run-time for minimizing the energy of the protein 2PTH with an upper bound of iterations is reduced by 28% and – much more importantly – the result is closer to the optimum.

Thus, with minimal effort in the application of our novel tool, we have been able to identify a problematic situation in a framework that has been in use since 1996, have been able to rectify it using higher precision, and have not only succeeded in improving the quality of the computation, but surprisingly also reduced the runtime.

6.6 GLPK

Linear programming solvers are prone to rounding errors and therefore are interesting targets for an analysis. We performed an analysis of the GNU Linear Programming Kit (GLPK) [7] with an integer linear problem mentioned by Neumaier and Shcherbina [17]

$$
\begin{array}{ll}
\min & -x_{20} \\
\text{s.t.} & (s+1)\,x_1 - x_2 \geqslant s-1, \\
& -s x_{i-1} + (s+1)\,x_i - x_{i+1} \geqslant (-1)^i\,(s+1) \\
& \qquad \text{for } i = 2:19, \\
& -s x_{18} - (3s-1)\,x_{19} + 3x_{20} \geqslant -(5s-7)\,, \\
& 0 \leqslant x_i \leqslant 10 \text{ for } i = 1:13, \\
& 0 \leqslant x_i \leqslant 10^6 \text{ for } i = 14:20, \\
& \text{all } x_i \text{ integers},
\end{array}
$$

The problem consists of 20 integer variables. For $s = 6$, the problem is solvable, but several solvers are unable to find the solution

$$x = (1, 2, 1, 2, \ldots, 1, 2)^T$$

GLPK in version 4.47 reports that the "problem has no integer feasible solution". The important constraint is

$$0 \leqslant x_i \leqslant 10^6 \text{ for } i = 14:20$$

For GLPK we found that the problem is solvable if 10^6 is replaced by a number between 2 and 21871. Therefore, we analyzed and compared the runs for the upper bounds 21871 and 21872.

For the upper bound 21871, our analysis reveals that -21871 is stored in a double precision value and occurs correctly in operations that are influenced by a tree of preceding operations with depth 159. As the largest tree of preceding operations measured has also a depth of 159, it is likely that this operation has an influence on the final result.

Increasing the upper bound by one to 21872 leads to noticeable changes in the result of the analysis. As before, -21872 is also stored in a double precision value and occurs in a operation that is influenced by a tree of depth 161. Here, 161 is the maximum depth of a tree of preceding operations. However, the original value is 0 and the shadow value is exactly -21872.

This shows that floating-point issues detected by our analysis lead to the wrong result of GLPK. Our analysis gives more details on the highly inaccurate value and automatically outputs an error trace for it, but a deeper analysis is beyond the scope of this paper.

7. Related Work

A manual rounding error analysis is one of the best ways to prove the stability of algorithms and to find errors because it works with a mathematical model of floating-point arithmetic. But because it has to be done by a human expert, it is only feasible for small programs.

Surprisingly few tools have been developed to assist the programmer in locating floating-point accuracy problems. Recent approaches are based on a static analysis (e.g. Fluctuat [9]) or try to dynamically detect or correct errors (e.g. automated error correction by Langlois [14]). A static analysis has the advantage that properties can be proven, however, often only with large error bounds. Furthermore, static approaches suffer from the incapability of disambiguating heap accesses as common in static analyses. Hence, such approaches only work well on scalar computations.

Brown et al. [2] designed an analysis to determine if floating-point operations can be optimized by going to a lower precision or by replacing floating-point with fixed-point arithmetic. This is of interest e.g. for synthesizing FPGAs. Their tool FloatWatch also builds on Valgrind. To employ fixed-point arithmetic, they track the overall range of all floating-point values. In addition, they track the maximum difference between single and double precision computations by performing all double precision operations side by side in single precision.

An et al. [1] present a dynamic binary analysis called FPInst based on DynInst to calculate errors side by side. In contrast to our analysis, the shadow values do not contain values in higher precision but an absolute error in double precision. Their functions for computing the error of instructions are derived from work by Dekker [4]. The formulas from Dekker enable higher precision floating-point arithmetic on top of lower precision arithmetic, but here the formulas are used to track the error throughout a program. Applied to the simple error accumulation shown in Section 2.2.2, the functions presented by An et al. give an error that alternates and does not monotonously increase like the real error. This shows that Dekker's formulas can not always be used to compute error accumulation and also explains the large discrepancies between the computed error and the real error in their examples.

Lam et al. [13] present a dynamic binary analysis based on DynInst to find floating-point operations where digits are canceled. This work can be seen as the one most similar to ours. Their analysis works by tracking all cancellations and reporting them. To minimize the output, the number of reports for the same instruction is decreased logarithmically. Finally, the number of cancellations per instruction and the average number of canceled bits per instruction are computed. However, the approach does not distinguish between benign and catastrophic cancellations, reducing its applicability in practice. Because most cancellations are benign, the user is left with many false positives.

8. Conclusions

In this paper, we presented a dynamic program analysis to detect floating-point accuracy problems. To our knowledge, it is the first dynamic analysis that detects catastrophic cancellations and uses a light-weight slicing approach at tracing accuracy problems through the program. We implemented our analysis in the Valgrind dynamic binary instrumentation framework and exercise it on large-scale benchmark programs in all of which we detect accuracy problems of varying severity: one problem slows down the convergence of an algorithm, other ones cause catastrophic cancellation leading to totally insignificant output of the problem. We showed how our analysis helps in tracking down and successfully fixing the causes of these issues.

9. Acknowledgments

This work is partly supported by the Intel Visual Computing Institute in Saarbrücken.

References

[1] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. FPInst: Floating point error analysis using dyninst, 2008. http://www.freearrow.com/downloads/files/fpinst.pdf.

[2] A. W. Brown, P. H. J. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, pages 6–16, 2007.

[3] S. P. E. Corporation. SPEC CPU2006 benchmarks. http://www.spec.org/cpu2006/.

[4] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971. 10.1007/BF01397083.

[5] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS '09*, pages 53–69. Springer-Verlag, 2009.

[6] S. Gal. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Trans. Math. Softw.*, 17:26–45, March 1991.

[7] GNU Linear Programming Kit, ver. 4.47. http://www.gnu.org/software/glpk/.

[8] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23:5–48, 1991.

[9] E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI'11*, pages 232–247, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition edition, 2002.

[11] A. Hildebrandt, A. K. Dehof, A. Rurainski, A. Bertsch, M. Schumann, N. Toussaint, A. Moll, D. Stockel, S. Nickels, S. Mueller, H.-P. Lenhof, and O. Kohlbacher. BALL - biochemical algorithms library 1.3. *BMC Bioinformatics*, 11(1):531, 2010.

[12] W. Kahan. How futile are mindless assessments of roundoff in floating-point computation?, 2006. http://www.cs.berkeley.edu/ wkahan/Mindless.pdf.

[13] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. In *WHIST 11*, 2011.

[14] P. Langlois. Automatic linear correction of rounding errors. *BIT Numerical Mathematics*, 41:515–539, 2001.

[15] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[16] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100. ACM, 2007.

[17] A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming*, 99:283–296, 2004. 10.1007/s10107-003-0433-3.

[18] U. G. A. Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia, GAO report IMTEC 92-26, 1992. http://www.gao.gov/products/IMTEC-92-26/.

[19] J. Walker. Floating-point benchmarks. http://www.fourmilab.ch/fbench/, retrieved on 2011-03-03.

[20] The Wall Street Journal November 8, 1983, p.37.

[21] V. Weaver. SPEC CPU2006 problems of valgrind. http://thread.gmane.org/gmane.comp.debugging.valgrind.devel/1488/, retrieved on 2011-03-03.