Techniques for the automatic debugging of scientific floating-point programs

David H. Bailey[1], James Demmel[2], William Kahan[2], Guillaume Revy[2], and Koushik Sen[2]

[1]Berkeley Lab Computing Sciences, Computational Research Division, Lawrence Berkeley National Laboratory
[2]Parallel Computing Laboratory, EECS Department, University of California at Berkeley

1. Context and purpose of our work

- Tool for automatically detecting and remedying anomalies in scientific floating-point programs
  - large-scale scientific single/multi-threaded applications has been growing rapidly
  - anomalies may cause rare but critical bugs that are hard for nonexperts to find or fix [1]
    ↪ detection and remedy either at C code level or at run-time

- **What are the usual anomalies?**
  - rounding error accumulations
  - conditional branches involving floating-point comparisons
    ↪ may go astray due to the subtleties of floating-point arithmetic, eg NaN
    ↪ convergence misbehavior
  - difficulties of programming languages
    ↪ Fortran: constants converted in full double precision accuracy if written with the `d--` notation, otherwise not, unlike C
  - under/overflows, resolution of ill-conditioned problems
    ↪ returned result may be completely wrong
  - cancellation, benign or catastrophic, ...

2. Usual approaches for finding anomalies in floating-point programs?

- **Some techniques for detecting these usual anomalies [1],[3]**
  - altering rounding mode of floating-point arithmetic hardware
    ↪ may not normally be usable to remedy the problems
  - extending precision of floating-point computation
    ↪ may increase run time significantly (due to the use of software interface)
  - using interval arithmetic
    ↪ produces a certificate, but run time cost is the greatest

- **Example of precision extension**

```c
int main(void)
{
  float a = 1e15f;
  float b = 1.0f;
  float c = a + b;
  float d = c - a;

  printf("The value of d is: %1.19e\n", d);
  return 0;
}
```

```
$ The value of d is: 0.0000000000000000000e+00
```
✗

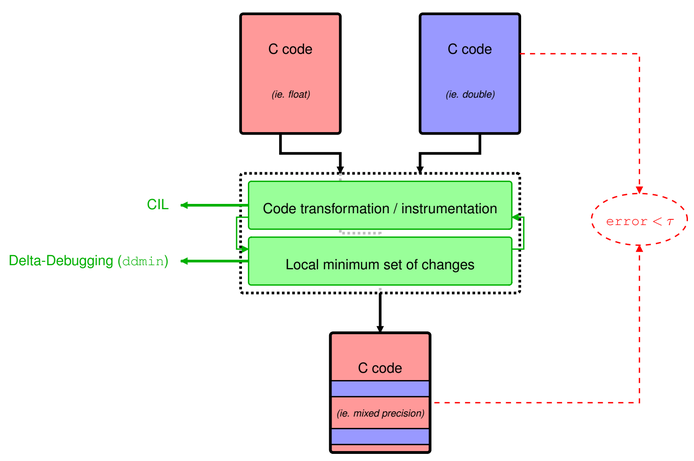
```c
int main(void)
{
  double a = 1e15f;
  double b = 1.0f;
  double c = a + b;
  double d = c - a;

  printf("The value of d is: %1.19e\n", d);
  return 0;
}
```

```
$ The value of d is: 1.0000000000000000000e+00
```
✓

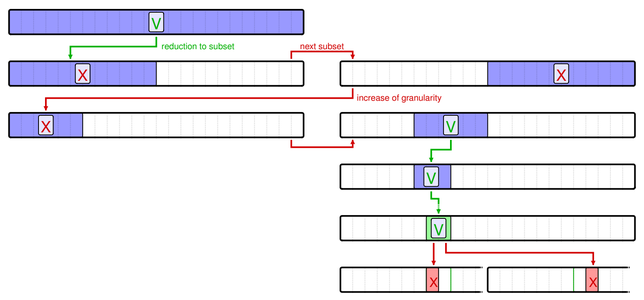
Only two variables (eg `b` and `c`) have to be declared in `double`. How to detect quickly the parts of a C program the most sensitive to given parameters?

3. Detection of anomalies using delta-debugging algorithm and code transformations

- **General flowchart of the framework**



- **Delta-debugging algorithm [5]**
  - General principle: find a local minimal set of changes on a given C code, so that the computed result remains within a given threshold of a known and more accurate result (exact, high precision, ...)
    ↪ implementation like binary search



  ↪ the computed local minimal subset of changes is 1-minimal

- **Code transformation and instrumentation done using CIL (C Intermediate Language) [4]**
  - analysis and source-to-source transformation of C programs
  - C program → tree structure: definition of transformations for each kind of node (variable declaration, constants, function definition, ...)

    ↪ `FloatToDouble`: float → double,
    ↪ `RoundingMode`: RN → {RU,RD,RZ},
    ↪ `FlipFunction`: flipping between two implementations of the same computation,
    ↪ `DoubleToDD`: double → double-double (implemented using QD package [2]).

Combinaison of delta-debugging and code transformations for finding areas of a C code the most sensitive to given parameters

4. Some examples

- **Inaccurate computation of the arc length of a given function [1]**

$$g(x) = x + \sum_{0 \le k \le 5} 2^{-k} \sin(2^k x), \quad \text{over } (0, \pi).$$

  - summing for $x_k \in (0, \pi)$ divided into n subintervals

$$\sqrt{h^2 + (g(x_k + h) - g(h))^2}, \quad \text{with } h = \pi/n \text{ and } x_k = kh.$$

For $n = 1000000$: sum $= 5.795776322412856$ (double-double) → 20 × slower
$= 5.795776322413031$ (double)
$= 5.795776322412856$ (double-double sum of doubles)

↪ only 1 change is necessary: found in ≈ 30 sec.

- **Bug in `dgges` subroutine of LAPACK**

I have the following problem with `dgges`. For version 3.1.1 and sooner, I get a reasonable result, for version 3.2 and 3.2.1, I get `info=n+2`.

  - the only difference between LAPACK 3.1.1 and 3.2.x: some call to `dlarfg` replaced by `dlarfp`

    ↪ which call(s) to `dlarfp` made the program fail?

Result obtained in ≈ 1 m. 50 sec.
- ▷ 25610 possible changes
- ▷ 34 (47) tests done
- ▷ all changes but 1 did not matter

5. Conclusion and future work

- **Current work on the automatic debugging of scientific floating-point applications**
  1. CIL for applying transformations on a given C code,
  2. delta-debugging algorithm for finding a minimal set of effective changes to be applied on a given C code to improve its accuracy.

- **Future work**
  - implementation of other transformations (eg FloatToFF: float to float-float)
  - application of these automated techniques to bug reports of widely used library (eg LAPACK), and automation of techniques that are originally done by hand
    ↪ behavior when NaNs are input or occur during the run
  - detection of some infinite loops, exception handling, ...
  - automatic and careful addition of an adjustable "fuzz" (small numerical value) on one side of the comparisons that go astray due to the subtleties of floating-point arithmetic
  - automatic user's program scanning and modification when a constant is not converted to full expected precision because of difficulties of the programming language

Some references

[1] David H. Bailey. Resolving Numerical Anomalies in Scientific Computation. 2008.
Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/numerical-bugs.pdf`.

[2] David H. Bailey, Yozo Hida, Xiaoye S. Li, and Brandon Thompson. QD – C++/Fortran-90 double-double and quad-double package.
Available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[3] William Kahan. How Futile are Mindless Assessments of Roundoff in Floating-point Computation? 2006.
Available at `http://www.eecs.berkeley.edu/~wkahan/Mindless.pdf`.

[4] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compiler Construction*, 2002.

[5] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.