Techniques for the automatic debugging of scientific floating-point programs

Guillaume Revy
ParLab EECS University of California, Berkeley

Motivation & Objective

- The field of large-scale scientific application has been growing rapidly
  ⇒ anomalies: significative impact on numerical results
  ⇒ on the general behavior of the systems

- Techniques for detecting anomalies vary:
  ⇒ in the costs of their application
  ⇒ and in the kind of anomalies they detect.
Motivation & Objective

- The field of large-scale scientific application has been growing rapidly
  - anomalies: significative impact on numerical results
  - on the general behavior of the systems

- Techniques for detecting anomalies vary:
  - in the costs of their application
  - and in the kind of anomalies they detect.

- Propose automatic techniques for detecting and remedying a wide class of numerical anomalies arising in single/multi-threaded applications
  - helping developers not necessarily expert in numerical analysis
  - improving their productivity
First simple example

Code

```c
#include <math.h>
#include <stdio.h>

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;
}
```

Execution result

$ The value of d is: 0.000000000000000000e+00$
First simple example

Code

```c
#include <math.h>
#include <stdio.h>

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;
}
```

Execution result

$ The value of d is: 1.0000000000000000e+00$
Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs
  - 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 \texttt{d__} notation, otherwise not, unlike C
  - under/overflows, resolution of ill-conditioned problems
    - returned result may be completely wrong
  - cancellation, benign or catastrophic, ...
Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs
  - either at C code level or at run-time

- How to detect these usual anomalies?
  - 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
    - intervals may grow too wide to be useful
Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs
  - either at C code level or at run-time

How to detect these usual anomalies?

- 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
  - intervals may grow too wide to be useful

How to detect quickly the most sensitive part of a C program?
Framework flowchart

C code

(ie. float)

C code

(ie. double)

Framework for debugging

floating-point programs

C code

(ie. mixed precision)

error $< \tau$
Framework flowchart

- C code (ie. float)
- C code (ie. double)
- Code transformation / instrumentation
- Local minimum set of changes
- Delta-Debugging (ddmin)
- CIL
- C code (ie. mixed precision)
- Error $\leq \tau$
Outline of the talk

1. Delta-Debugging Algorithm

2. Code transformation and instrumentation

3. Some results

4. Conclusion & Current work
Outline of the talk

1. Delta-Debugging Algorithm

2. Code transformation and instrumentation

3. Some results

4. Conclusion & Current work
General principle of Delta-Debugging

- **Principle:** find a *local minimal* set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)
  - implementation like binary search
General principle of Delta-Debugging

- **Principle**: find a local minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)
  - implementation like binary search
General principle of Delta-Debugging

- **Principle**: find a **local minimal** set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

  → implementation like binary search
General principle of Delta-Debugging

- **Principle**: find a local minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

  → implementation like binary search
General principle of Delta-Debugging

- **Principle**: find a **local minimal** set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

  → implementation like binary search
General principle of Delta-Debugging

- **Principle:** find a local minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

  → implementation like binary search
General principle of Delta-Debugging

- Principle: find a local minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)
  - implementation like binary search
General principle of Delta-Debugging

- **Principle**: find a *local minimal* set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

  → implementation like binary search
General principle of Delta-Debugging

- **Principle**: find a local minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)
  
  → implementation like binary search
General principle of Delta-Debugging

Principle: find a **local minimal** set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

→ implementation like binary search
Delta-Debugging Algorithm for the first simple example

Code

```c
#include <math.h>
#include <stdio.h>

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

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

- 13 possible changes
- 7 (9) tests done
- 2 changes are relevant

Execution result

$ The value of d is: 1.0000000000000000000e+00$
Delta-Debugging Algorithm

Let \( \text{error}, \ C_\nu = S_1 \cup \cdots \cup S_n \), and \( \bar{S}_i \) be such that:

\[
\text{error}(\emptyset) = x, \quad \text{error}(C_\nu) = \checkmark, \quad \text{and} \quad \bar{S}_i = C_\nu - S_i.
\]

Finally \( \text{ddmin}(C_\nu) = \text{DD}(C_\nu, 2) \) with

1. if \( \exists i \in \{1, \cdots, n\} \) such that \( \text{error}(S_i) = \checkmark \)
   \( \rightarrow \) reduction to subset: \( \text{DD}(S_i, 2) \),

2. if \( \exists i \in \{1, \cdots, n\} \) such that \( \text{error}(\bar{S}_i) = \checkmark \)
   \( \rightarrow \) reduction to complement: \( \text{DD}(\bar{S}_i, \max(n - 1, 2)) \),

3. if \( n < |C_\nu| \)
   \( \rightarrow \) increase of granularity: \( \text{DD}(C_\nu, \min(|C_\nu|, 2n)) \),

4. otherwise
   \( \rightarrow \) done.
Delta-Debugging Algorithm

Let $\text{error}$, $C_\nu = S_1 \cup \cdots \cup S_n$, and $\bar{S}_i$ be such that:

$$\begin{align*}
\text{error}(\emptyset) &\geq \tau, \\
\text{error}(C_\nu) &< \tau, \\
\text{and} &\\
\bar{S}_i &\equiv C_\nu - S_i.
\end{align*}$$

Finally $\text{ddmin}(C_\nu) = \text{DD}(C_\nu, 2)$ with

1. if $\exists i \in \{1, \cdots, n\}$ such that $\text{error}(S_i) < \tau$
   \[
   \rightarrow \text{ reduction to subset: } \text{DD}(S_i, 2),
   \]

2. if $\exists i \in \{1, \cdots, n\}$ such that $\text{error}(\bar{S}_i) < \tau$
   \[
   \rightarrow \text{ reduction to complement: } \text{DD}(\bar{S}_i, \max(n-1, 2)),
   \]

3. if $n < |C_\nu|$
   \[
   \rightarrow \text{ increase of granularity: } \text{DD}(C_\nu, \min(|C_\nu|, 2n)),
   \]

4. otherwise
   \[
   \rightarrow \text{ done.}
   \]
**Property on $\text{ddmin}$**

**Property**

*For any $S_i \subseteq C_\nu$, $\text{ddmin}(S_i)$ is 1-minimal.*
Property on $\text{ddmin}$

**Property**

For any $S_i \subseteq C$, $\text{ddmin}(S_i)$ is $1$-minimal.
Property on $\text{ddmin}$

Property

For any $S_i \subset C_v$, $\text{ddmin}(S_i)$ is $1$-minimal.
Outline of the talk

1. Delta-Debugging Algorithm

2. Code transformation and instrumentation

3. Some results

4. Conclusion & Current work
CIL - **C Intermediate Language**

- **CIL**: high-level representation of C programs
  - analysis and source-to-source transformation of C programs

- C program: represented as a tree
  - a node = variable declaration, constants, function definition, block statement, ...
  - scan in depth-first the structure of the CIL program (tree)
  - define modifications (transformations) on each kind of node
CIL - C Intermediate Language

- CIL: high-level representation of C programs
  ⇒ analysis and source-to-source transformation of C programs

- C program: represented as a tree
  ⇒ a node = variable declaration, constants, function definition, block statement, ...
  ⇒ scan in depth-first the structure of the CIL program (tree)
  ⇒ define modifications (transformations) on each kind of node

C code transformations using CIL

+ Local minimal set finding using Delta-Debugging
Currently implemented transformations

- **FloatToDouble**: $\text{float} \rightarrow \text{double}$,
- **RoundingMode**: $\text{RN} \rightarrow \{\text{RU}, \text{RD}, \text{RZ}\}$,
- **FlipFunction**: flipping between two implementations of the same computation,
- **DoubleToDD**: $\text{double} \rightarrow \text{double-double}$ (Grey Ballard’s CS 263 project).
Outline of the talk

1. Delta-Debugging Algorithm

2. Code transformation and instrumentation

3. Some results

4. Conclusion & Current work
More realistic example (D.H. Bailey)

Problem

Calculate the arc length of the function $g$:

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

Solution

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

$$\sqrt{h^2 + (g(x_k + h) - g(h))^2},$$

with $h = \pi / n$ and $x_k = kh$. If $n = 1000000$, we have

result $\approx 5.795776322412856$ (double-double) $\rightarrow$ 20x slower

$\approx 5.795776322413031$ (double)

$\approx 5.795776322412856$ (double-double sum of doubles)
More realistic example (D.H. Bailey)

Solution

Summing for \( x_k \in (0, \pi) \) divided into \( n \) subintervals

\[
\sqrt{h^2 + (g(x_k + h) - g(h))^2},
\]

with \( h = \pi/n \) and \( x_k = kh \). If \( n = 1000000 \), we have

\[
\begin{align*}
\text{result} & = 5.795776322412856 \quad \text{(double-double)} \rightarrow 20x \text{ slower} \\
& = 5.795776322413031 \quad \text{(double)} \\
& = 5.795776322412856 \quad \text{(double-double sum of doubles)}
\end{align*}
\]

Automation with Delta-Debugging

- 57 possible changes
- 10 (10) tests done \( \approx 30 \) sec.
- only 1 change is necessary
Bug in \texttt{dgges} subroutine of LAPACK

**Bug report**

*I have the following problem with \texttt{dgges}. For version 3.1.1 and sooner, I get a reasonable result, for version 3.2 and 3.2.1, I get \textit{info}=n+2.*

- The only difference between LAPACK 3.1.1 and 3.2.x
  - some call to \texttt{dlarfg} replaced by \texttt{dlarfp}

- Which call(s) to \texttt{dlarfp} made the program fail?

**Automation with Delta-Debugging**

- 25610 possible changes
- 34 (47) tests done \(\approx 1 \text{ m. 50 sec.}\)
- all changes but 1 did not matter
Outline of the talk

1. Delta-Debugging Algorithm
2. Code transformation and instrumentation
3. Some results
4. Conclusion & Current work
Conclusion & Current work

- Framework for the **automatic debugging** of floating-point programs: detecting and remedying of a wide range of numerical anomalies
  - transformation / instrumentation using CIL
  - effective changes found using Delta-Debugging

- Delta-Debugging Algorithm
  - 1-minimality is not enough (in our cases)
  - how to determine initial set of changes?
  - implementation of other transformations (FloatToFF, ...)
  - protect some parts of code

- Adding an adjustable “fuzz” on one side of the comparisons that go astray

- Detection of some infinite loops, exception handling, ...