Open Source Quality (OSQ) Retreat

Santa Cruz, CA, USA, May 13, 2010

Techniques for the automatic debugging of scientific floating-point programs

Guillaume Revy

ParLab EECS University of California, Berkeley

Joint work with David H. Bailey, James Demmel, William Kahan, and Koushik Sen.

Motivation & Objective

- The field of large-scale scientific application has been growing rapidly
  - $\Rightarrow$ anomalies: significative impact on numerical results
  - $\Rightarrow$ on the general behavior of the systems
- Techniques for detecting anomalies vary:
  - $\Rightarrow$ in the costs of their application
  - $\Rightarrow$ 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

```
#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: 0.0000000000000000000e+00
```

First simple example

Code

```
#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: 1.0000000000000000000e+00
```

Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs

  $\longrightarrow$ either at C code level or at run-time

- What are the usual anomalies?
  - rounding error accumulations
  - conditional branches involving floating-point comparisons

    $\longrightarrow$ may go astray due to the subtleties of floating-point arithmetic, eg NaN

    $\longrightarrow$ convergence misbehavior
  - difficulties of programming languages

    $\longrightarrow$ 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

    $\longrightarrow$ 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

  $\longrightarrow$ either at C code level or at run-time

- How to detect these usual anomalies?
  - altering rounding mode of floating-point arithmetic hardware

    $\longrightarrow$ may not normally be usable to remedy the problems
  - extending precision of floating-point computation

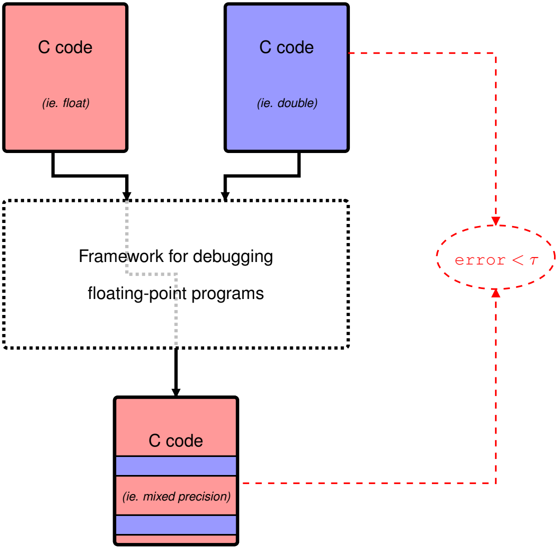
    $\longrightarrow$ may increase run time significantly (due to the use of software interface)
  - using interval arithmetic

    $\longrightarrow$ produces a certificate, but run time cost is the greatest

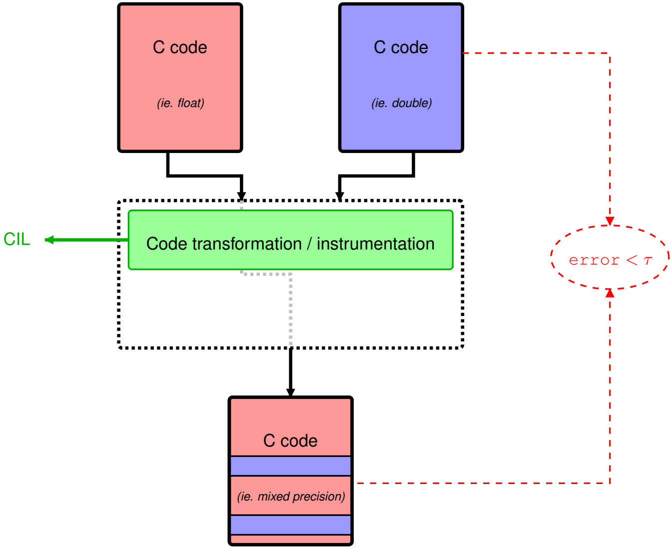
    $\longrightarrow$ intervals may grow too wide to be useful

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

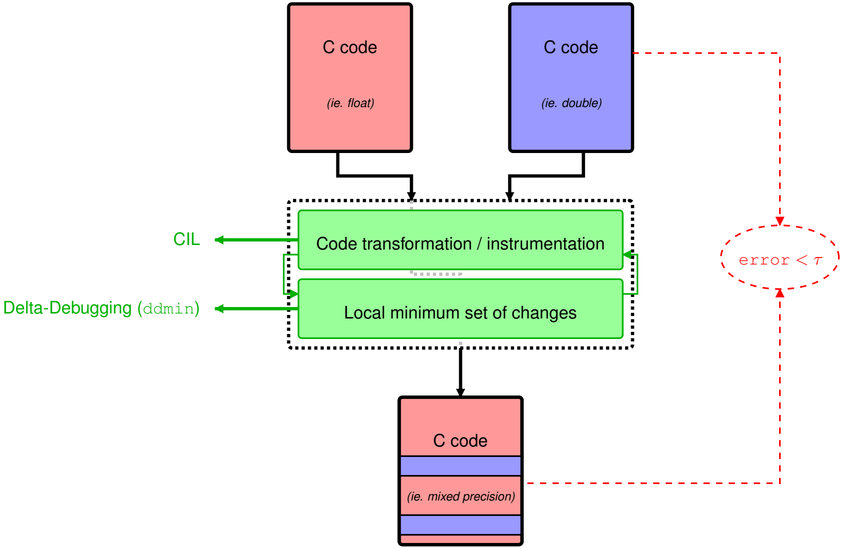
Framework flowchart



Framework flowchart



Framework flowchart



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

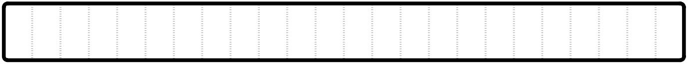
3. Some results

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, ...)

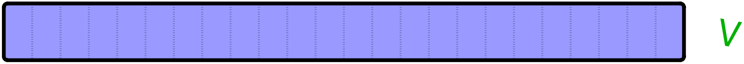
  $\longrightarrow$ implementation like binary search



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, ...)

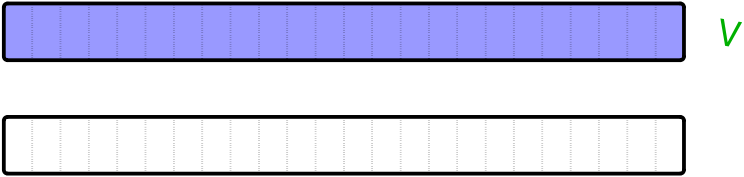
  $\longrightarrow$ implementation like binary search



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, ...)

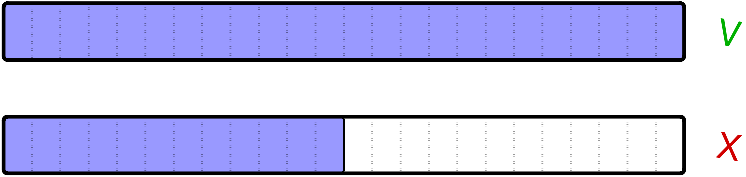
  → implementation like binary search



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, ...)

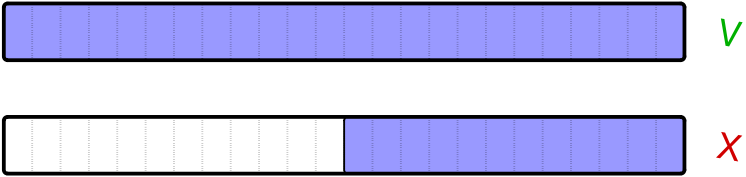
  → implementation like binary search



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, ...)

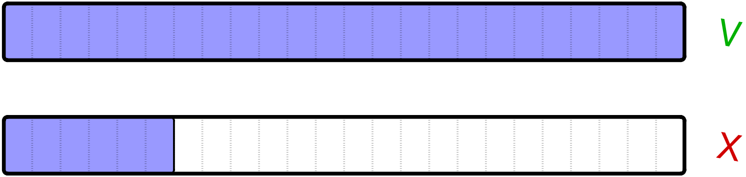
  $\longrightarrow$ implementation like binary search



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, ...)

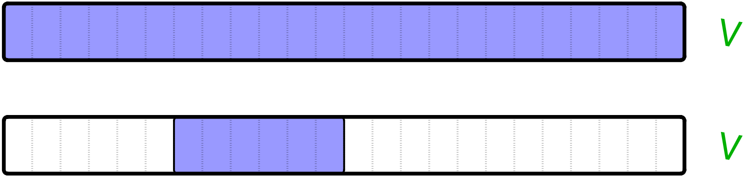
  $\longrightarrow$ implementation like binary search



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, ...)

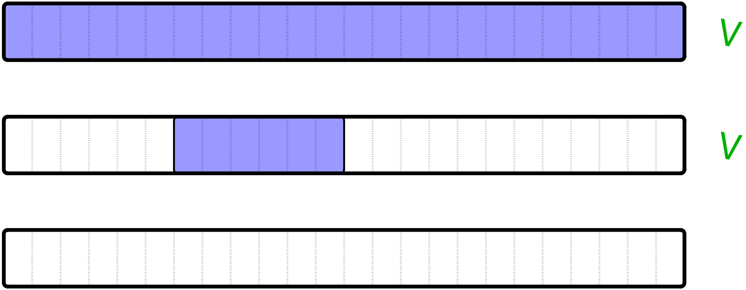
  → implementation like binary search



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, ...)

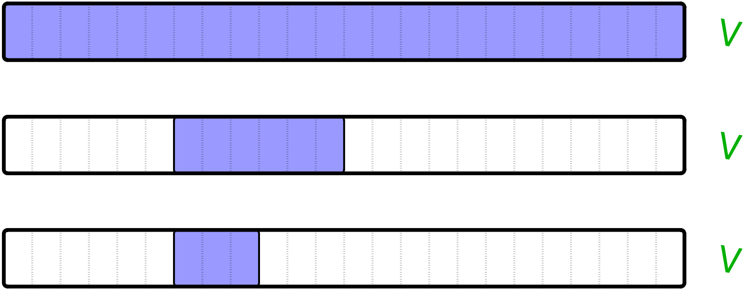
  → implementation like binary search



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, ...)

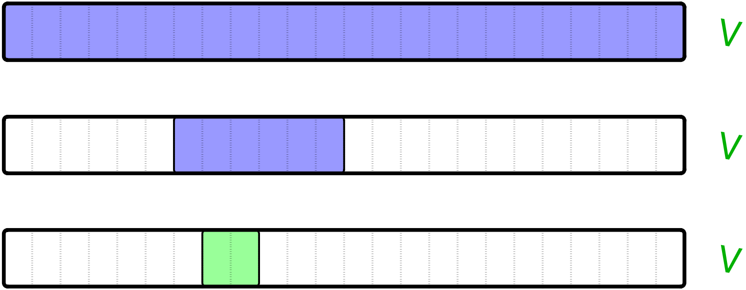
  → implementation like binary search



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

```
#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 $\texttt{error}$, $C_{\checkmark} = S_1 \cup \cdots \cup S_n$, and $\bar{S}_i$ be such that:

$$\texttt{error}(\emptyset) = \text{✗}, \quad \texttt{error}(C_{\checkmark}) = \checkmark, \quad \text{and} \quad \bar{S}_i = C_{\checkmark} - S_i.$$

Finally $\texttt{ddmin}(C_{\checkmark}) = \texttt{DD}(C_{\checkmark}, 2)$ with

1. if $\exists i \in \{1, \cdots, n\}$ such that $\texttt{error}(S_i) = \checkmark$
   $\longrightarrow$ reduction to subset: $\texttt{DD}(S_i, 2)$,
2. if $\exists i \in \{1, \cdots, n\}$ such that $\texttt{error}(\bar{S}_i) = \checkmark$
   $\longrightarrow$ reduction to complement: $\texttt{DD}(\bar{S}_i, \max(n-1, 2))$,
3. if $n < |C_{\checkmark}|$
   $\longrightarrow$ increase of granularity: $\texttt{DD}(C_{\checkmark}, \min(|C_{\checkmark}|, 2n))$,
4. otherwise
   $\longrightarrow$ done.

Delta-Debugging Algorithm

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

$$\texttt{error}(\emptyset) \geq \tau, \quad \texttt{error}(C_{\checkmark}) < \tau, \quad \text{and} \quad \bar{S}_i = C_{\checkmark} - S_i.$$

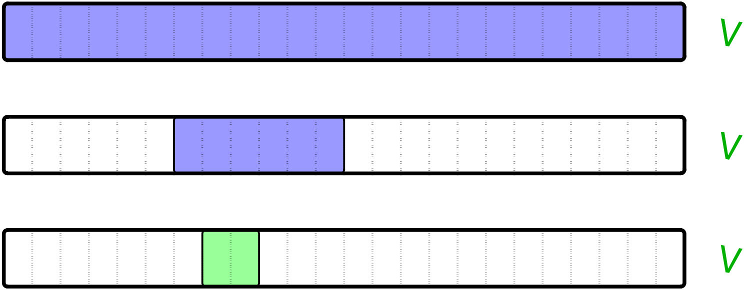
Finally $\texttt{ddmin}(C_{\checkmark}) = \texttt{DD}(C_{\checkmark}, 2)$ with

1. if $\exists i \in \{1, \cdots, n\}$ such that $\texttt{error}(S_i) < \tau$
   $\longrightarrow$ reduction to subset: $\texttt{DD}(S_i, 2)$,
2. if $\exists i \in \{1, \cdots, n\}$ such that $\texttt{error}(\bar{S}_i) < \tau$
   $\longrightarrow$ reduction to complement: $\texttt{DD}(\bar{S}_i, \max(n-1, 2))$,
3. if $n < |C_{\checkmark}|$
   $\longrightarrow$ increase of granularity: $\texttt{DD}(C_{\checkmark}, \min(|C_{\checkmark}|, 2n))$,
4. otherwise
   $\longrightarrow$ done.

Property on `ddmin`

Property

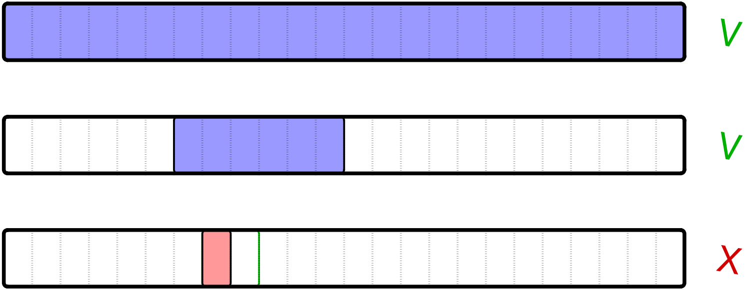
For any $S_i \subset C_{\checkmark}$, $\mathtt{ddmin}(S_i)$ is 1-minimal.



Property on `ddmin`

Property

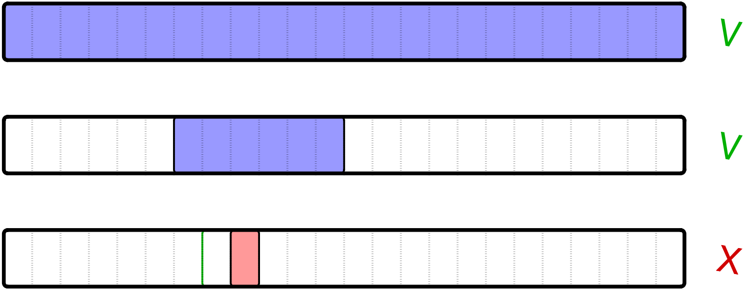
For any $S_i \subset C_{\checkmark}$, $\mathtt{ddmin}(S_i)$ is 1-minimal.



Property on ddmin

Property

For any $S_i \subset C_{\checkmark}$, $\mathtt{ddmin}(S_i)$ 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** **I**ntermediate **L**anguage

- 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** **I**ntermediate **L**anguage

- 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`: float $\longrightarrow$ double,
- `RoundingMode`: RN $\longrightarrow$ {RU,RD,RZ},
- `FlipFunction`: flipping between two implementations of the same computation,
- `DoubleToDD`: double $\longrightarrow$ 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:

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

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	=	5.795776322412856	(double-double) → 20x slower
	=	5.795776322413031	(double)
	=	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

$$\begin{aligned} \text{result} &= 5.795776322412856 \quad \text{(double-double)} \rightarrow 20x \text{ slower} \\ &= 5.795776322413031 \quad \text{(double)} \\ &= 5.795776322412856 \quad \text{(double-double sum of doubles)} \end{aligned}$$

Automation with Delta-Debugging

- 57 possible changes
- 10 (10) tests done $\approx$ 30 sec.
- only 1 change is necessary

Bug in dgges subroutine of LAPACK

Bug report

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
  - $\longrightarrow$ some call to `dlarfg` replaced by `dlarfp`
- Which call(s) to `dlarfp` made the program fail?

Automation with Delta-Debugging

- ▷ 25610 possible changes
- ▷ 34 (47) tests done $\approx$ 1 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, ...