Techniques for the automatic debugging of scientific floating-point programs

David H. Bailey[1] James Demmel[2] William Kahan[2]
Guillaume Revy[3] Koushik Sen[2]

Berkeley Lab Computing Sciences[1] ParLab (EECS, University of California, Berkeley)[2] DALI project-team - UPVD/LIRMM (CNRS-UM2)[3]

Thanks to Sun/Oracle.

Outline of the talk

1. Motivation and objective of the project

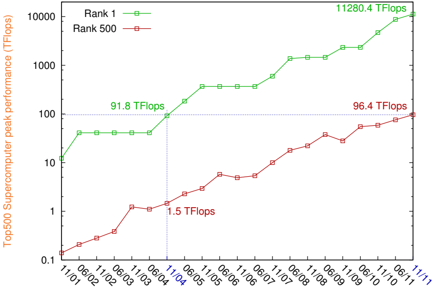
2. Locating numerical anomalies

3. Conclusion and perspective

Outline of the talk

Motivation and objective

- The field of large-scale scientific applications has been growing rapidly
  - in cycles used



  - in complexity of software

  ⇝ both these make finding bugs harder (especially for non experts)

- Objectives of the project
  - reduce difficulty of debugging
  - automatic techniques for detecting and suggesting remedies for roundoff and other numerical exception problems (anomalies)

Motivation and objective

- Techniques for detecting suspected anomalies vary :
  1. in the costs of their application,
  2. and in scopes and effectiveness : in the kind of anomalies they detect.

Motivation and objective

- Techniques for detecting suspected anomalies vary :
  1. in the costs of their application,
  2. and in scopes and effectiveness : in the kind of anomalies they detect.

- Goal $\rightsquigarrow$ automate debugging now done by hand
  - intelligent and automatic tool
    - locate automatically suspected anomalies
    - with or without source code
    - at runtime or statically
  - help developers whose expertise does not extend to numerical error-analysis
    - shorten debugging time and improve their productivity

What are the usual source of anomalies ?

- Common source of anomalies :
  - rounding error accumulations
  - conditional branches involving floating-point comparisons
    - e.g. NaN leading to a convergence misbehavior
  - vagaries of programming languages
    - Fortran : conversion of constants in full double precision accuracy (unlike C)

`double precision c1 0.1d0`	⇝	0.10000000000000000555
`double precision c2 0.1`	⇝	0.10000000149011611938

  - under/overflows
  - cancellation, benign or catastrophic
  - resolution of ill-conditioned problems, ...

- Hardware problems : misbehavior of floating-point programs

Example of avoidable numerical anomalies

Given n, evaluate the definite integral using Simpson's rule, with $h = \frac{b-a}{2 \cdot n}$

$$\int_a^b f(x) \cdot dx = \frac{h}{3}\Big[f(a) + 4 \cdot f(a+h) + 2 \cdot f(a+2 \cdot h) + \cdots + 4 \cdot f\big(a + (2 \cdot n - 1) \cdot h\big) + f(b)\Big].$$

```
unsigned int n = 10000;
double a = 0, b = 1, h = (b-a)/(2.*n);

double xk = a;          // xk ≈ a + k.n
double r = f(xk);       // r ≈ f(a)

while( 1 ){
  xk = xk + h;
  r = r + 4.* f(xk);    // r ≈ r + 4.f(a + k.n)

  xk = xk + h;
  if (xk >= b)
    break;

  r = r + 2.* f(xk);    // r ≈ r + 2.f(a + k.n)
}

r = r + f(xk);          // r ≈ r + f(b)

r = (h/3.) * r;
```

- Implementation with a conditional branch
  - terminate the loop when $a + k \cdot h \geq b$
  - may result in an extra iteration, if $a + k \cdot h$ is slightly less than b due to roundoff error

Example of avoidable numerical anomalies

Given n, evaluate the definite integral using Simpson's rule, with $h = \frac{b-a}{2 \cdot n}$

$$\int_a^b f(x) \cdot dx = \frac{h}{3} \left[f(a) + 4 \cdot f(a+h) + 2 \cdot f(a + 2 \cdot h) + \cdots + 4 \cdot f\big(a + (2 \cdot n - 1) \cdot h\big) + f(b) \right].$$

```
unsigned int n = 10000;
double a = 0, b = 1, h = (b-a)/(2.*n);

double xk = a;          // xk ≈ a + k.n
double r = f(xk);       // r ≈ f(a)

while( 1 ){
  xk = xk + h;
  r = r + 4.* f(xk);    // r ≈ r + 4.f(a + k.n)

  xk = xk + h;
  if (xk + EPSILON >= b)
    break;

  r = r + 2.* f(xk);    // r ≈ r + 2.f(a + k.n)
}

r = r + f(xk);          // r ≈ r + f(b)

r = (h/3.) * r;
```

- Implementation with a conditional branch
  - terminate the loop when $a + k \cdot h \geq b$
  - may result in an extra iteration, if $a + k \cdot h$ is slightly less than b due to roundoff error
- Example of fix : $a + k \cdot h + \varepsilon \geq b$
  - how to determine ε semi-automatically ?

Example of avoidable numerical anomalies

Given n, evaluate the definite integral using Simpson's rule, with $h = \frac{b-a}{2\cdot n}$

$$\int_a^b f(x)\cdot dx = \frac{h}{3}\left[f(a)+4\cdot f(a+h)+2\cdot f(a+2\cdot h)+\cdots+4\cdot f\big(a+(2\cdot n-1)\cdot h\big)+f(b)\right].$$

```
unsigned int n = 10000;
double a = 0, b = 1, h = (b-a)/(2.*n);

double xk = a;          // xk ≈ a + k.n
double r = f(xk);       // r ≈ f(a)

while( 1 ){
  xk = xk + h;
  r = r + 4.* f(xk);    // r ≈ r + 4.f(a + k.n)

  xk = xk + h;
  if (xk + EPSILON >= b)
    break;

  r = r + 2.* f(xk);    // r ≈ r + 2.f(a + k.n)
}

r = r + f(xk);          // r ≈ r + f(b)

r = (h/3.) * r;
```

- Implementation with a conditional branch
  - terminate the loop when $a+k\cdot h \geq b$
  - may result in an extra iteration, if $a+k\cdot h$ is slightly less than b due to roundoff error
- Example of fix : $a+k\cdot h+\varepsilon \geq b$
  - how to determine ε semi-automatically ?
- Another fix : make loop variable integer

Outline of the talk

1. Motivation and objective of the project

2. Locating numerical anomalies

3. Conclusion and perspective

How to detect these usual anomalies ?

- Detection can be done by static or dynamic analysis
- When suspected, these usual anomalies may be detected by :
  - altering rounding mode of floating-point arithmetic hardware
    - ⇝ may not be available on every architectures (e.g. GPU)
  - extending precision of floating-point computation
    - ⇝ may increase runtime significantly (use of software implementation)
  - modifying comparisons by adding an unobvious tolerance
  - using interval arithmetic
    - ⇝ produces a certificate, but runtime cost increases significantly
    - ⇝ intervals may grow too wide to be useful
  - using Error-Free Transformation (EFT)
    - ⇝ code transformations may be difficult to automate

How to detect these usual anomalies ?

- Detection can be done by static or dynamic analysis
- When suspected, these usual anomalies may be detected by :
  - altering rounding mode of floating-point arithmetic hardware
    - ⇝ may not be available on every architectures (e.g. GPU)
  - extending precision of floating-point computation
    - ⇝ may increase runtime significantly (use of software implementation)
  - modifying comparisons by adding an unobvious tolerance
  - using interval arithmetic
    - ⇝ produces a certificate, but runtime cost increases significantly
    - ⇝ intervals may grow too wide to be useful
  - using Error-Free Transformation (EFT)
    - ⇝ code transformations may be difficult to automate

How to detect these usual anomalies ?

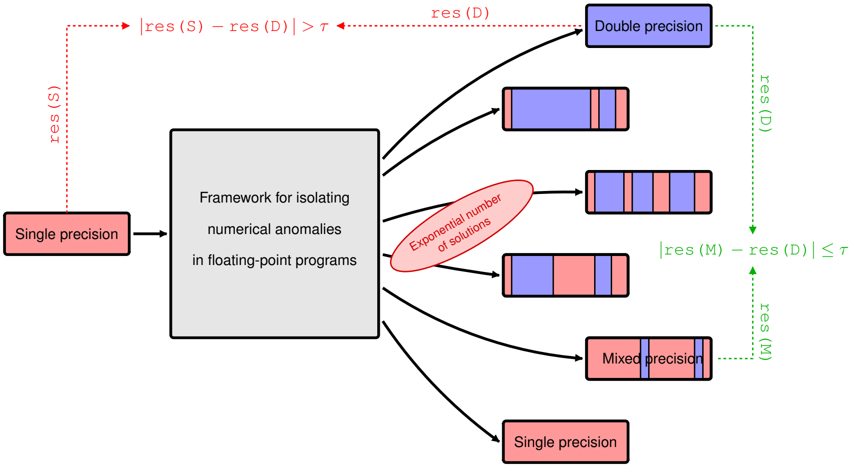
- Detection can be done by static or dynamic analysis
- When suspected, these usual anomalies may be detected by :
  - altering rounding mode of floating-point arithmetic hardware
    - ⇝ may not be available on every architectures (e.g. GPU)
  - extending precision of floating-point computation
    - ⇝ may increase runtime significantly (use of software implementation)
  - modifying comparisons by adding an unobvious tolerance
  - using interval arithmetic
    - ⇝ produces a certificate, but runtime cost increases significantly
    - ⇝ intervals may grow too wide to be useful
  - using Error-Free Transformation (EFT)
    - ⇝ code transformations may be difficult to automate

How to quickly detect usual anomalies using dynamic analysis ?

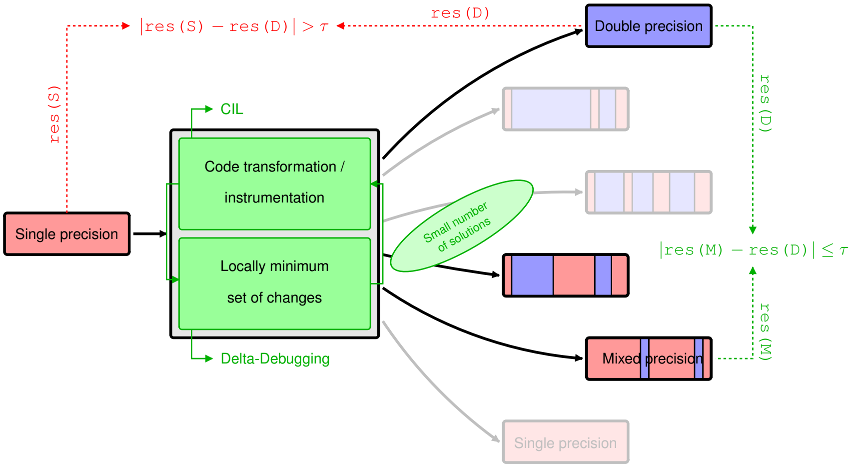
Isolating failure-inducing code fragment

- Does the anomaly (bug) really depend on the whole input code ?
- First step in processing any bug : simplification
  - eliminating all the details in the original code that are not relevant
    - $\rightsquigarrow$ isolate the difference that causes the bug (failure)
  - often people spend a lot of time isolating failure-inducing code fragment
- Usually carried out by hand
  - long and tedious + may miss some relevant simplification
  - need to automate this process

Framework flowchart



Framework flowchart



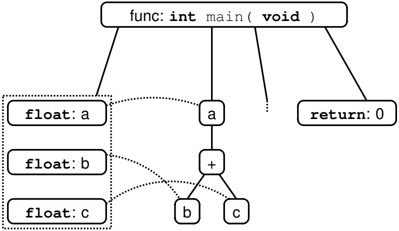
Transformations using CIL (**C** **I**ntermediate **L**anguage)

- CIL [1] : high-level representation of C programs
  - analysis and source-to-source transformation of C programs

```
int main( void )
{
  float a;
  float b;
  float c;

  a = b + c;
  // ...

  return 0;
}
```



1. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs. Proc. of Conference on Compiler Construction, 2002

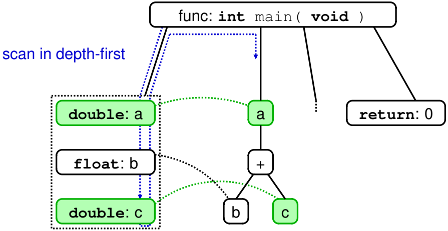
Transformations using CIL (C Intermediate Language)

- CIL [1] : high-level representation of C programs
  - analysis and source-to-source transformation of C programs

```
int main( void )
{
  double a;
  float b;
  double c;

  a = b + c;
  // ...

  return 0;
}
```



1. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs. Proc. of Conference on Compiler Construction, 2002

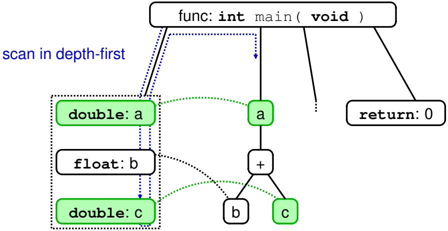
Transformations using CIL (C Intermediate Language)

- CIL [1] : high-level representation of C programs
  - analysis and source-to-source transformation of C programs

```
int main( void )
{
  double a;
  float b;
  double c;

  a = b + c;
  // ...

  return 0;
}
```

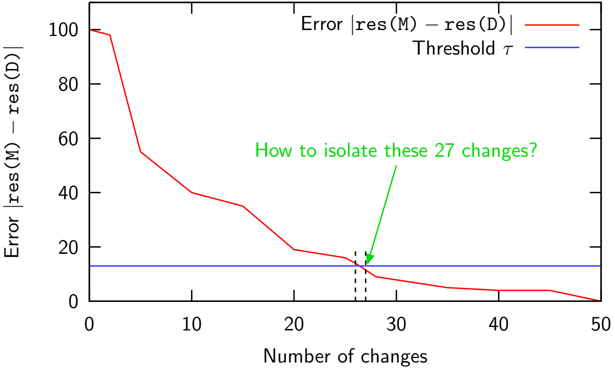


- Currently implemented transformations
  - `FloatToDouble` : float ⇒ double,
  - `RoundingMode` : RN ⇒ {RU,RD,RZ},
  - `DoubleToDD` : double ⇒ double-double,
  - `FlipFunction` : implementation1 ⇒ implementation2.

1. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs. Proc. of Conference on Compiler Construction, 2002

Delta-Debugging algorithm

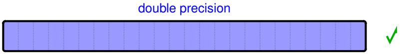
- General principle of Delta-Debugging [2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)



2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

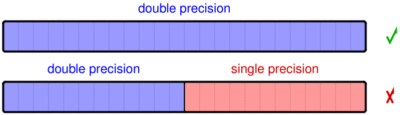
- General principle of Delta-Debugging[2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search



2. A. Zeller and R. Hildebrandt.
Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging[2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search

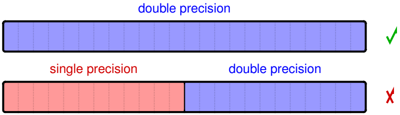


$\rightsquigarrow$ apply half the changes and check if the output is still accurate

2. A. Zeller and R. Hildebrandt.
Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging [2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search

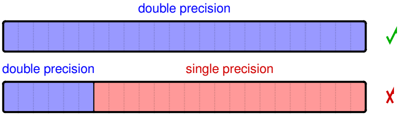


- ⇝ apply half the changes and check if the output is still accurate
- ⇝ if no, go back to the other state and discard the other half

2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging [2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search

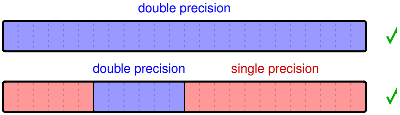


- ⇝ apply half the changes and check if the output is still accurate
- ⇝ if no, go back to the other state and discard the other half
- ⇝ if the input is still inaccurate, increase the granularity of the splitting

2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging[2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search

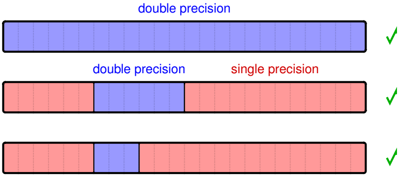


- ⇝ apply half the changes and check if the output is still accurate
- ⇝ if no, go back to the other state and discard the other half
- ⇝ if the input is still inaccurate, increase the granularity of the splitting

2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging[2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search

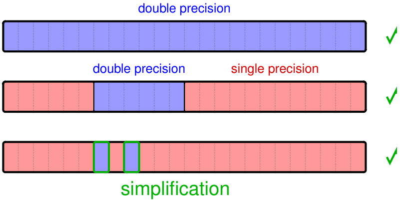


- ⇝ apply half the changes and check if the output is still accurate
- ⇝ if no, go back to the other state and discard the other half
- ⇝ if the input is still inaccurate, increase the granularity of the splitting

2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging[2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search



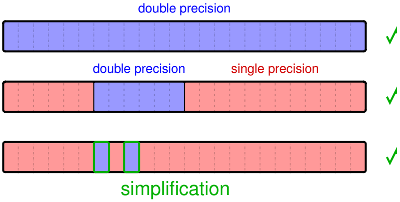
- ⇝ apply half the changes and check if the output is still accurate
- ⇝ if no, go back to the other state and discard the other half
- ⇝ if the input is still inaccurate, increase the granularity of the splitting

  - the changes may be not consecutive

2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

Delta-Debugging algorithm

- General principle of Delta-Debugging [2] : find a locally minimal set of changes on a code, so that the returned result remains within a given threshold τ of a more accurate result (exact, higher precision, ...)
  - implementation like binary search



- ⇝ apply half the changes and check if the output is still accurate
- ⇝ if no, go back to the other state and discard the other half
- ⇝ if the input is still inaccurate, increase the granularity of the splitting

  - the changes may be not consecutive

- Future improvement : consider the efficiency (performance) of changes ?

2. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002

More realistic example (D.H. Bailey)

- Calculate the arc length of the function g defined as

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

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

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

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

$$\begin{aligned} \text{result} &= 5.795776322412856 \quad \text{(all double-double)} \rightsquigarrow 20x \text{ slower} \\ &= 5.795776322413031 \quad \text{(all double)} \end{aligned}$$

More realistic example (D.H. Bailey)

- Calculate the arc length of the function g defined as

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

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

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

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

$$\begin{aligned}
\text{result} &= 5.795776322412\textcolor{blue}{2856} && \text{(all double-double)} \rightsquigarrow \textcolor{orange}{20x \text{ slower}} \\
&= 5.795776322413\textcolor{orange}{3031} && \text{(all double)} \\
&= 5.795776322412\textcolor{blue}{2856} && \text{(only the summand is in double-double)} \\
& && \rightsquigarrow \textcolor{orange}{\text{almost the same speed}}
\end{aligned}$$

  - only 1 change is necessary $\rightsquigarrow$ found in $\approx$ 30 sec.

Bug in `dgges` subroutine of LAPACK

LAPACK bug report [3]

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 calls to `dlarfg` replaced by `dlarfp`
- First step : simplification
  - which call(s) to `dlarfp` made the program fail ?

3. See `http://icl.cs.utk.edu/lapack-forum/viewtopic.php?f=2&t=1783` for details.

Bug in `dgges` subroutine of LAPACK

LAPACK bug report [3]

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 calls to `dlarfg` replaced by `dlarfp`
- First step : simplification
  - which call(s) to `dlarfp` made the program fail ?
- Automation with delta-debugging
  - 25610 calls to `dlarfp` = 25610 possible changes
  - all changes but 1 did not matter $\rightsquigarrow$ found in about 1m. 50 sec.
    - $\rightsquigarrow$ much easier to find which line of code was the source of this bug

3. See `http://icl.cs.utk.edu/lapack-forum/viewtopic.php?f=2&t=1783` for details.

Outline of the talk

NaN/Inf checking and some infinite loop isolation

- Problem : NaN/Inf occurring in floating-point programs may be the source of some infinite loops

- How to detect where NaN/Inf occur during the run of the program ?
  - by testing the result of each floating-point operations
  - by testing overflow/invalid flag (only works if created by programs, not input)
    - $\rightsquigarrow$ need an exception handler

NaN/Inf checking and some infinite loop isolation

- Problem : NaN/Inf occurring in floating-point programs may be the source of some infinite loops
- How to detect where NaN/Inf occur during the run of the program ?
  - by testing the result of each floating-point operations
  - by testing overflow/invalid flag (only works if created by programs, not input)
    - $\rightsquigarrow$ need an exception handler
- Just try to show loop termination or detect loop non-termination

Conclusion and perspective

- Goal $\rightsquigarrow$ automatic debugging of scientific floating-point programs
- Framework for the automatic isolation of numerical anomalies
  - transformation / instrumentation using CIL
  - effective changes found using Delta-Debugging
- Current and future work
  - implementation of other transformations (FloatToFF, ...)
  - apply to database bugs of LAPACK
  - NaN/Inf checking for isolating some infinite loops (loop non-termination)
  - modifying comparisons that go astray by adding an unobvious tolerance
  - isolation of floating-point constants that are not converted in full precision
  - identification of hardware features that could facilitate “debugging process” while not impacting normal floating-point performance

Centre National d'Études Spatiales (CNES) - Séminaire CCT/STIL
Toulouse, France, January 19, 2012

Techniques for the automatic debugging of scientific floating-point programs

David H. Bailey[1] James Demmel[2] William Kahan[2]
Guillaume Revy[3] Koushik Sen[2]

Berkeley Lab Computing Sciences[1] ParLab (EECS, University of California, Berkeley)[2] DALI project-team - UPVD/LIRMM (CNRS-UM2)[3]

Thanks to Sun/Oracle.