Automated Repair of High Inaccuracies in Numerical Programs

Xin Yi, Liqian Chen, Xiaoguang Mao, Tao Ji
National University of Defense Technology, China

09/22/2017 ICSME 2017
Introduction

- High-inaccuracy bug
  - An input $x$
  - Real arithmetic output $O_r(x)$ (i.e., mathematical output)
  - Floating-point arithmetic output $O_f(x)$
  - Threshold $\varepsilon$

\[
\left| \frac{O_r(x) - O_f(x)}{O_r(x)} \right| > \varepsilon
\]
Introduction

Rounding error
Introduction

• Hard to debug and fix high-inaccuracy bugs manually
  • Huge search space (input domain)
    • More than $9.0e+15$ floating-point (64 bits) numbers in [1,2]
  • Hard to localize the buggy code
    • Propagation and accumulation of round errors
  • Need of special knowledge on floating-point arithmetic to modify the buggy code
Introduction

Automated repair of numerical program:

Detecting + Localizing + Repairing
High-inaccuracy bugs
Our Approach

Four phases for automated repair

Detecting High-inaccuracy Bugs

Localizing Buggy Code

Generating and Validating Patches

Patch Application and Simplification
Example

double F(double x){
    //assert(-10<x<100);
    double y,d,z;
    z = 0.0;
    if (x > 0.0){
        x = pow(x,5);
        y = x-1.0;
    }
    else{
        d = x*x;
        y = d-1.0;
    }
    while(z < 1e10){
        z = x*x-y*y;
        x = x*2.0+1.0;
    }
    y = y*z;
    return y;
}

Input intervals
• $I_1$: $[-10.0, 0.0)$
• $I_2$: $[0.0, 100.0]$
Our Approach

Phase 1: Detecting High-inaccuracy Bugs

• Obtaining (approximate) mathematical output
  • Shadow value execution in higher precision (64bits to 128 bits) (FPDebug) [Benz ’12]

• Detecting algorithm to find negative test cases
  • Locality-Sensitive Genetic Algorithm (LSGA) [Zou ’15]
  • Binary Guided Random Testing (BGRT) [Chiang ’14]
Our Approach

Phase 1: Detecting High-inaccuracy Bugs

• Using FPDebug to approximate the real arithmetic results and Binary Guided Random Testing to search inputs.

Input intervals triggering bugs
• $I_1: x \in [-1.0042, -0.9982]$
• $I_2: x \in [39.5303, 100.0000]$
Our Approach

Phase 2: Localizing buggy code

control flow graph  \rightarrow  Slices and Blocks

Input intervals triggering bugs
- $I_1: x \in [-1.0042, -0.9982]$
- $I_2: x \in [39.5303, 100.0000]$
Our Approach

Phase 2: Localizing buggy code

• Ranking blocks according to the relative error that each block introduces

Slices and Blocks

Ranking list of Blocks

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b-1</td>
<td>b-2</td>
<td></td>
</tr>
<tr>
<td>b-2</td>
<td>b-3</td>
<td></td>
</tr>
<tr>
<td>b-3</td>
<td>b-1</td>
<td></td>
</tr>
</tbody>
</table>

### Our Approach

**Phase 2: Localizing buggy code**

- Ranking blocks according to the relative error that each block introduces.
Our Approach

Phase 3: Generating and Validating Patches

- Generating patches
  - symbolical calculation
  - mathematically equivalent transformation

\[ d = x \times x \]
\[ y = d - 1.0 \]

\[ d = x \times x \]
\[ y = x \times x - 1 \]

\[ d = x \times x \]
\[ y = (x - 1) \times (x + 1) \]
Our Approach

Phase 3: Generating and Validating Patches

- Validating Patches
- Regression testing

\[ \text{Input intervals trigger bugs} \]
- \( I_1: x \in [-1.0042, -0.9982] \)
- \( I_2: x \in [39.5303, 100.0000] \)

\begin{verbatim}
Our Approach
if ((x>= -1.0042) && (x<=-0.9982)){
d = x*x;
y = d-1.0;
}
else{
d = x*x;
y = d-1.0;
}

while(z < 1e10){
z = x*x-y*y;
x = x*2.0+1.0;
}
while(z < 1e10){
z = (x-y)*(x+y);
x = x*2.0+1.0;
}
\end{verbatim}
Our Approach

Phase 4: Patch Application

double F(double x){
    //assert(-10<x<100);
    double y,d,z;
    z = 0.0;
    if (x > 0.0){
        x = pow(x,5);
        y = x-1.0;
    }
    else{
        d = x*x;
        y = d-1.0;
    }
    while(z < 1e10){
        z = x*x-y*y;
        x = x*2.0+1.0;
    }
    y = y*z;
    return y;
}

double F(double x){
    //assert(-10<x<100);
    double y,d,z;
    z = 0.0;
    if (x > 0.0){
        x = pow(x,5);
        y = x-1.0;
    }
    else{
        if ((x>= -1.0042) &&(x<-0.9982)){
            d = x*x;
            y = (x-1.0)*(x+1.0);
        }else{
            d = x*x;
            y = d-1.0;
        }
    }
    if ((x>=35.5303) &&(x<=100)){
        while(z<1e10){
            z = (x-y)*(x+y);
            x = x*2.0+1.0;
        }
        while(z < 1e10){
            z = x*x-y*y;
            x = x*2.0+1.0;
        }
        y = y*z;
        return y;
    }
Our Approach

Phase 4: Patch Simplification

def F(x):
    """
    //assert(-10<x<100);
    double y,d,z;
    z = 0.0;
    if (x > 0.0){
        x = pow(x,5);
        y = x-1.0;
    }
    else{
        d = x*x;
        y = (x-1.0)*(x+1.0);
    }
    while(z < 1e10){
        z = (x-y)*(x+y);
        x = x*2.0+1.0;
    }
    y = y*z;
    return y;
    """
Our Approach

Before repair

Max error > 1e-13

Max error > 8e-7

After repair

Max error < 1e-15

Max error < 1e-15
## Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Domain</th>
<th>Time(s)</th>
<th>Max. Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time for Detecting</td>
<td>Time for Patches</td>
</tr>
<tr>
<td>frac2</td>
<td>(0,1e5]</td>
<td>120.22</td>
<td>5.06</td>
</tr>
<tr>
<td>frac3</td>
<td>(1,200]</td>
<td>75.54</td>
<td>14.87</td>
</tr>
<tr>
<td>sqrt2</td>
<td>(0,1e7]</td>
<td>123.71</td>
<td>5.04</td>
</tr>
<tr>
<td>sqrt2</td>
<td>(0,1e10]</td>
<td>217.94</td>
<td>3.11</td>
</tr>
<tr>
<td>rsqrt</td>
<td>(0,700]</td>
<td>93.76</td>
<td>9.58</td>
</tr>
</tbody>
</table>

Benchmark: 5 programs from FPBench (a benchmark for floating point analysis [Damouche '16])
Conclusion

• Propose a novel approach for automatically detecting, localizing, and repairing high-inaccuracy bugs in numerical programs

• Develop an automated repair prototype tool, evaluate it on several benchmark programs and achieve promising results
Future Work

- Design more efficient detecting algorithm to find negative test cases
- Improve our tool and evaluate it on real-world scientific numerical programs, e.g., the GNU Scientific Library (GSL)
Thank you!