ANALYSIS AND SYNTHESIS OF FLOATING-POINT Routines

Zvonimir Rakamarić
ROADMAP

1. Introduction
2. Floating-point Research at Utah
3. Floating-points in SMACK Verifier
4. Floating-point Error Analysis
5. Dynamic Analysis
6. Static Analysis
7. Synthesis
INTRODUCTION
FP COMPUTATIONS ARE UBIQUITOUS
IEEE 754 STANDARD

- Well-known floating-point standard
- Published in 1985
- Almost everyone follows it

- So why are we even talking about this?
CHALLENGES

- FP is “weird”
  - Does not faithfully match math (finite precision)
  - Non-associative
  - Heterogeneous hardware support
- FP code is hard to get right
  - Lack of good understanding
  - Lack of good and extensive tool support
- FP software is large and complex
  - High-performance computing (HPC) simulations
  - Stock exchange
FP IS WEIRD

- Finite precision and rounding
  - \( x + y \) in reals \( \neq \) \( x + y \) in floating-point
- Non-associative
  - \( (x + y) + z \neq x + (y + z) \)
  - Creates issues with
    - Compiler optimizations (e.g., vectorization)
    - Concurrency (e.g., reductions)
- Standard completely specifies only \(+, -, *, /,\) comparison, remainder, and square root
  - Only recommendation for some functions
    (trigonometry)
Heterogeneous hardware support
- $x + y^*z$ on Xeon $\neq x + y^*z$ on Xeon Phi
  - Fused multiply-add
- Intel’s online article “Differences in Floating-Point Arithmetic Between Intel Xeon Processors and the Intel Xeon Phi Coprocessor”

Common sense does not (always) work
- $x$ “is better than” $\log(e^x)$
- $(e^x - 1)/x$ “can be worse than” $(e^x - 1)/\log(e^x)$
  - Error cancellation
HARD TO GET RIGHT

- Writing a simple triangle classifier is challenging
- Poor (no?) tool support in practice

Pascal Cuoq on John Regehr’s blog:
“The problem with floating-point is that people start with a vague overconfident intuition of what should work, and progressively refine this intuition by removing belief when they are bitten by implementations not doing what they expected.”
HARD TO GET RIGHT cont.

- Uintah HPC framework developers
  - Advanced, senior, knowledgeable developers
  - Tedious manual debugging to root-cause a floating-point-related bug
- Personal communication (paraphrasing)
  - “When I turned on vectorization my output suddenly changed.”
  - “My OpenMP program occasionally returns a different output.”
  - “I have no idea what is going on.”
REAL-WORLD EXAMPLES OF BUGS

- Patriot missile failure in 1991 (webpage)
  - Miscalculated distance due to floating-point error
  - Time in tenths of second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds
- Inconsistent FP calculations in Uintah

```
P/C = 161.9999…
floor(P/C) = 161

Xeon

Expecting 161 msgs

Sent 162 msgs

Xeon Phi

P/C = 162
floor(P/C) = 162
```
FLOATING-POINT NUMBERS

- Sign, mantissa, exponent:
  \[ ((-1)^S) \times 1.M \times 2^E \]
- Single precision: 1, 23, 8
- Double precision: 1, 52, 11
FLOATING-POINT NUMBER LINE

- 3 bits for precision
- Between any two powers of 2, there are $2^3 = 8$ representable numbers
ROUNDING IS SOURCE OF ERRORS

Real Numbers

64-bit FP

\[(\tilde{x} - x) (\tilde{y} - y)\]
ROUNDING MODES

- 4 standard rounding modes
  - Round to nearest (default)
  - Round to 0
  - Round to plus infinity
  - Round to minus infinity

- Can be controlled
  - For experts only
**ERROR GROWS WITH MAGNITUDE**

Table 2-13  Gaps Between Representable Single-Format Floating-Point Numbers

<table>
<thead>
<tr>
<th>x</th>
<th>nextafter(x, +∞)</th>
<th>Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.4012985e−45</td>
<td>1.4012985e−45</td>
</tr>
<tr>
<td>1.1754944e−38</td>
<td>1.1754945e−38</td>
<td>1.4012985e−45</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0000001</td>
<td>1.1920929e−07</td>
</tr>
<tr>
<td>2.0</td>
<td>2.0000002</td>
<td>2.3841858e−07</td>
</tr>
<tr>
<td>16.000000</td>
<td>16.000002</td>
<td>1.9073486e−06</td>
</tr>
<tr>
<td>128.00000</td>
<td>128.00002</td>
<td>1.5258789e−05</td>
</tr>
<tr>
<td>1.0000000e+20</td>
<td>1.0000001e+20</td>
<td>8.7960930e+12</td>
</tr>
<tr>
<td>9.9999997e+37</td>
<td>1.0000001e+38</td>
<td>1.0141205e+31</td>
</tr>
</tbody>
</table>
FLOATING-POINT OPERATIONS

- First normalize to the same exponent
  - Smaller exponent -> shift mantissa right
- Then perform the operation
- Losing bits when exponents are not the same!
UTAH FLOATING-POINT TEAM

1. Ganesh Gopalakrishnan (prof)
2. Zvonimir Rakamarić (prof)
3. Alexey Solovyev (alumni postdoc)
4. Wei-Fan Chiang (alumni PhD)
5. Ian Briggs (staff programmer)
6. Mark Baranowski (MS)
7. Dietrich Geisler (alumni undergrad)
8. Liam Machado (undergrad)
9. Rocco Salvia (PhD)
**RESEARCH THRUSTS**

**Analysis**
1. Verification of floating-point programs
2. Estimation of floating-point errors
   1. Dynamic
      - Best effort
      - Produces lower bound (under-approximation)
   2. Static
      - Rigorous
      - Produces upper bound (over-approximation)

**Synthesis**
1. Rigorous mixed-precision tuning
SMACK VERIFIER

http://smackers.github.io
FLOATING-POINTS IN SMACK

- Support verification of properties that require precise reasoning about floating-points
- Leverage floating-point decision procedures implemented in Satisfiability Modulo Theories (SMT) solvers
  - Z3 SMT solver for now
- Stable version released
  - Enables verification of floating-point programs in C
- Drive research on better decision procedures by providing benchmarks for SMT
ERROR ANALYSIS
FLOATING-POINT ERROR

Input values: x, y

Finite precision
\[ z_{fp} = f_{fp}(x, y) \]

Infinite precision
\[ z_{inf} = f_{inf}(x, y) \]

Absolute error:
\[ | z_{fp} - z_{inf} | \]

Relative error:
\[ \left| \frac{z_{fp} - z_{inf}}{z_{inf}} \right| \]
ERROR PLOT FOR MULTIPLICATION

Absolute Error

X values

Y values

X axis

Y axis
ERROR PLOT FOR ADDITION

Absolute Error

X values

Y values

X axis

Y axis
USAGE SCENARIOS

- Reason about floating-point computations
- Precisely characterize floating-point behavior of libraries
- Support performance-precision tuning and synthesis
- Help decide where error-compensation is needed
- “Equivalence” checking
DYNAMIC ANALYSIS

http://github.com/soarlab/S3FP

Efficient Search for Inputs Causing High Floating-point Errors, PPoPP 2014
GOAL

- Finding program inputs that maximize floating-point error
TESTING FOR FP ERRORS

inputs
X0 ← •
X1 ← •
X2 ← •

Low precision program → Low precision result

Error calculation

High precision program → High precision result
**MAIN INSIGHT**

- Random testing with good guidance heuristics can outperform naïve random testing.
- We propose search-based random testing for maximizing floating-point error.

![Diagram showing the comparison between Naïve random testing, Search-based random testing, Max. error, and Over approximation.](image)
CONFIDENTIALITY

- An assignment from input variables to intervals

\[
\begin{align*}
\text{configuration} & : \\
X0 & \leftarrow [0.0, 1.0] \\
X1 & \leftarrow [1.1, 2.2] \\
X2 & \leftarrow [2.3, 3.3]
\end{align*}
\]
Our Approach

- Sample inputs to find *sour-spots* causing high floating-point error on program output.

![Diagram](image-url)

Program

- **configuration**
  - \(X_0 \leftarrow [\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet] \)
  - \(X_0 = 0.5\)
  - \(X_1 = 1.5\)
  - \(X_2 = 3.0\)

- **X0**: 0.0 0.5 1.0
- **X1**: 1.1 1.5 2.2
- **X2**: 2.3 3.0 3.3

High floating-point error
GENETIC-BASED ALGORITHM

- Starting config.
- Generate candidate sub-configurations
- Candidates: \{ sub-conf. 1 \} \ldots \{ sub-conf. n \}
- Restart?
- Choose best sub-conf.
- Program

or

sub-conf. k best among candidates
SUMMARY

- Guided testing overcomes some drawbacks of previous approaches
  - Improves scalability to real codes
  - Precisely handles diverse floating-point operations and conditionals

- Guided testing can detect (much) higher floating-point errors than pure random testing
Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions, FM 2015

http://github.com/soarlab/FPTaylor
CONTRIBUTIONS

- Handles non-linear and transcendental functions
- Tight error upper bounds
  - Better than previous work
- Rigorous
  - Over-approximation
  - Based on our own rigorous global optimizer
  - Emits a HOL-Lite proof certificate
    - Verification of the certificate guarantees estimate
- Tool called FPTaylor publicly available
Given FP Expression and Input Intervals

Obtain Symbolic Taylor Form

Obtain Error Function

Maximize the Error Function

Generate Certificate in HOL-Lite
IEEE ROUNDING MODEL

Consider $op(x, y)$ where $x$ and $y$ are floating-point values, and $op$ is a function from floats to reals.

IEEE round-off errors are specified as

$$op(x, y) \cdot (1 + e_{op}) + d_{op}$$

For normal values

For subnormal values

Only one of $e_{op}$ or $d_{op}$ is non-zero:

$$|e_{op}| \leq 2^{-24}, \quad |d_{op}| \leq 2^{-150}$$
ERROR ESTIMATION EXAMPLE

- Model floating-point computation of $E = x / (x + y)$ using reals as

$$
\tilde{E} = \frac{x}{(x + y) \cdot (1 + e_1) \cdot (1 + e_2)}
$$

- Absolute rounding error is then $|\tilde{E} - E|$
- We have to find the max of this function over
  - Input variables $x, y$
    - Exponential in the number of inputs
  - Additional variables $e_1, e_2$ for operators
    - Exponential in floating-point routine size!
SYMBOLIC TAYLOR EXPANSION

- Reduces dimensionality of the optimization problem
- Basic idea
  - Treat each $e$ as “noise” (error) variables
  - Now expand based on Taylor’s theorem
    - Coefficients are symbolic
    - Coefficients weigh the “noise” correctly and are correlated
- Apply global optimization on reduced problem
  - Our own parallel rigorous global optimizer called Gelpia
  - Non-linear reals, transcendental functions
ERROR ESTIMATION EXAMPLE

\[ \tilde{E} = \frac{x}{(x + y) \cdot (1 + e_1)} \cdot (1 + e_2) \]

expands into

\[ \tilde{E} = E + \frac{\partial \tilde{E}}{\partial e_1}(0) \times e_1 + \frac{\partial \tilde{E}}{\partial e_2}(0) \times e_2 + M_2 \]

where \( M_2 \) summarizes the second and higher order error terms and \( |e_0| \leq \epsilon_0, |e_1| \leq \epsilon_1 \)

Floating-point error is then bounded by

\[ |\tilde{E} - E| \leq \left| \frac{\partial \tilde{E}}{\partial e_1}(0) \right| \times \epsilon_1 + \left| \frac{\partial \tilde{E}}{\partial e_2}(0) \right| \times \epsilon_2 + M_2 \]
ERROR ESTIMATION EXAMPLE

- Using global optimization find constant bounds
- $M_2$ can be easily over-approximated
- Greatly reduced problem dimensionality
  - Search only over inputs $x, y$ using our Gelpia optimizer

\[ \forall x, y. \quad \left| \frac{\partial \tilde{E}}{\partial e_1} (0) \right| = \left| \frac{x}{x+y} \right| \leq U_1 \]

\[ |\tilde{E} - E| \leq \left| \frac{\partial \tilde{E}}{\partial e_1} (0) \right| \times \epsilon_1 + \left| \frac{\partial \tilde{E}}{\partial e_2} (0) \right| \times \epsilon_2 + M_2 \]
Operations are single-precision (32 bits)

\[ |\tilde{E} - E| \leq U_1 \times \epsilon_{32\text{-bit}} + U_2 \times \epsilon_{32\text{-bit}} \]

Operations are double-precision (64 bits)

\[ |\tilde{E} - E| \leq U_1 \times \epsilon_{64\text{-bit}} + U_2 \times \epsilon_{64\text{-bit}} \]
RESULTS FOR JETENGINE

jetEngine, $x_1 \in [-5, 5]$, $x_2 \in [-20, 5]$, Double Precision
SUMMARY

- New method for rigorous floating-point round-off error estimation
- Our method is embodied in new tool FPTaylor
- FPTaylor performs well and returns tighter bounds than previous approaches
SYNTHESIS

http://github.com/soarlab/FPTuner

Rigorous Floating-point Mixed-precision Tuning, POPL 2017
MIXED-PRECISION TUNING

Goal:
Given a real-valued expression and output error bound, automatically synthesize precision allocation for operations and variables
**APPROACH**

- Replace machine epsilons with symbolic variables
  \[ s_0, s_1 \in \{ \epsilon_{32-bit}, \epsilon_{64-bit} \} \]
  \[
  |\tilde{E} - E| \leq U_1 \times s_1 + U_2 \times s_2
  \]

- Compute precision allocation that satisfies given error bound
  - Take care of type casts

- Implemented in FPTuner tool
FPTuner TOOLFLOW

Routine: Real-valued Expression

Generic Error Model

Efficiency Model

Optimization Problem

Optimal Mixed-precision

User Specifications:
- Error Threshold
- Operator Weights
- Extra Constraints

Gelpia Global Optimizer

Gurobi
EXAMPLE: JACOBI METHOD

- Inputs:
  - 2x2 matrix
  - Vector of size 2
- Error bound: 1e-14
- Available precisions: single, double, quad
- FPTuner automatically allocates precisions for all variables and operations
PERFORMANCE BENEFITS

Graph showing the elapse time of mixed-precision versions (ns) versus the elapse time of all-128 versions (ns). The data points are scattered on a logarithmic scale, with a trend line indicating a correlation between the two measurements.
ENERGY CONSUMPTION BENEFITS

(a) sine
(b) sine
(c) maxBolt
(d) gaussian
(e) jetEngine
(f) reduction
SUMMARY

- Support mixed-precision allocation
- Based on rigorous formal reasoning
- Encoded as an optimization problem
- Extensive empirical evaluation
  - Includes real-world energy measurements showing benefits of precision tuning
CONCLUSIONS

- Verification of floating-point programs
  - Implemented in SMACK software verifier
  - Uses SMT, bit-precise
- Estimation of floating-point errors
  - Dynamic based on guided testing
  - Static based on Taylor expansion and global optimization
- Mixed-precision tuning
  - Leverages static error estimation to select optimal precision for each operation