When Does a Bit Matter? Techniques for Verifying the Correctness of Assembly Languages and Floating-Point Programs

Samuel D. Pollard

UNIVERSITY OF OREGON
Computer and Information Science

28 May 2021
1. Introduction

2. Binary Analysis

3. A Statistical Analysis of Error in MPI Reduction Operations

4. Scalable Error Analysis for Floating-Point Program Optimization

5. Conclusion and Future Research Directions
I enjoy working with either *no* abstraction or *lots* of abstraction

- Assembly 😊
- Java 😞
- Matlab 😊

I noticed a couple common abstractions which when they failed were hard to fix

- Instruction Set Architectures (ISAs)
- Floating Point (FP)
Some Intuitive Definitions

- *High-level*: using abstractions; not concerned with underlying implementation of a program
- *Low-level*: the opposite

**Key Challenge**

Abstractions give insight into the nature of a program.
How can we apply high-level reasoning techniques about computer programs to low-level implementations? Specifically,

1. How can we write specifications of instruction set architectures (ISAs) that enable static analysis for program verification?

2. How can we formalize and quantify the error from floating-point arithmetic in high-performance numerical programs?
1 Introduction

2 Binary Analysis

3 A Statistical Analysis of Error in MPI Reduction Operations

4 Scalable Error Analysis for Floating-Point Program Optimization

5 Conclusion and Future Research Directions
Quameleon: A Lifter and Intermediate Language for Binary Analysis

Based on previously published work in collaboration with Philip Johnson-Freyd, Jon Aytac, Tristan Duckworth, Michael J. Carson, Geoffrey C. Hulette, and Christopher B. Harrison [6]
Motivation

Need to analyze binaries on old, obscure ISAs
- ISAs not supported by existing tools
- No machine-readable specification
- Bad old days: No IEEE 754 floats, no 8-bit bytes

Other tools gain lots of efficiency from expressive ISAs and feature-rich Intermediate Languages (ILs)

We instead require an adaptable IL

Fun example: cLEMENCy ISA invented for DEFCON had 9-bit bytes, 27-bit words, middle-endian [9]
Architectural Overview

- ISA Specification DSL
- M6800
- Other ISAs
- Quameleon Intermediate Language
- Optimizations for Analysis
- Concrete Execution Engine
  - Custom Symbolic Execution Engines
  - Weakest Precondition
  - LLVM/KLEE
  - Angr toolchain (Symbolic Execution, etc.)
  - Abstract Interpretation
Architectural Overview

Quameleon Intermediate Language

- ISA Specification DSL
- M6800
- Other ISAs

- Optimizations for Analysis

- Concrete Execution Engine
  - Custom Symbolic Execution Engines
  - Weakest Precondition
  - LLVM/KLEE
  - Angr toolchain (Symbolic Execution, etc.)
  - Abstract Interpretation
Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries
- Lift binaries into a simple IL amenable to multiple analysis backends
- Close to LLVM IR in spirit
- Size of QIL (~ 60 instructions) means easy to manipulate, harder to write
- Balance this with Haskell as a macro-assembler for QIL
Quameleon Intermediate Language (QIL)

- Static Single Assignment (SSA)
- Program consists of a list of blocks, single entry, multiple-exit
- Data are stored in bit vectors of parametrizable width
- Can read/write to locations like RAM, registers
- Keep track of I/O as sequence of reads/writes
Haskell Embedded Domain Specific Language (DSL)
Sample M6800

A ← 0xE
A ← A & [0x40]

We want to match the manual precisely
... and Its Corresponding Semantics

\textbf{AND} r l \rightarrow \textbf{do}
\begin{align*}
\text{ra} & \leftarrow \text{getRegVal} \ r \\
\text{op} & \leftarrow \text{loc8ToVal} \ l \quad \text{--- Loc. of 8 bits in RAM} \\
\text{rv} & \leftarrow \text{andBit} \ ra \ op \\
\text{z} & \leftarrow \text{isZero} \ rv \\
\text{writeReg} \ r \ rv \\
\text{writeCC Zero} \ z \quad \text{--- CC = Condition Code} \\
\text{branch next}
\end{align*}
Back-ends

- HPCL
- Quameleon
- Intermediate Language
- M6800
- Optimizations for Analysis
- ISA Specification DSL

- Concrete Execution Engine
- Custom Symbolic Execution Engines
- Weakest Precondition
- LLVM/KLEE
- Angr toolchain (Symbolic Execution, etc.)
- Abstract Interpretation

Other ISAs
Current Back-ends

1. Emulator
2. Bridge to angr
   - angr is a symbolic execution engine primarily for cybersecurity
   - Treat QIL as an ISA that angr can execute
Optimizations

- ISA Specification DSL
- M6800
- Other ISAs
- Quameleon Intermediate Language
- Concrete Execution Engine
  - Custom Symbolic Execution Engines
  - Weakest Precondition
  - LLVM/KLEE
  - Angr toolchain (Symbolic Execution, etc.)
  - Abstract Interpretation
- Optimizations for Analysis
QIL-QIL Optimizations

The goal is to facilitate analysis

- Constant folding
- Branch to known value
- Dead code elimination
- Inlining with simple heuristics e.g., inline everywhere
- Defunctionalization

\[ \text{Reduce code size} \]
\[ \text{Simplify CFG} \]
Dissertation Question

How can we apply high-level reasoning techniques about computer programs to low-level implementations? Specifically,

1. How can we write specifications of instruction set architectures (ISAs) that enable static analysis for program verification?

2. How can we formalize and quantify the error from floating-point arithmetic in high-performance numerical programs?
1. Introduction

2. Binary Analysis

3. A Statistical Analysis of Error in MPI Reduction Operations

4. Scalable Error Analysis for Floating-Point Program Optimization

5. Conclusion and Future Research Directions
A Statistical Analysis of Error in MPI Reduction Operations

Based off previously published work with Boyana Norris [7].
A Brief Introduction to Floating-Point Arithmetic

The rest of this talk focuses on floating-point (FP) arithmetic and floating-point operations (FLOPs)

Ariane V, the $500 million overflow
We Don’t Trust Floating Point

- Doesn’t map perfectly to real numbers
- Can’t even represent 1/10 exactly
- Complex behavior of error and exceptions

[1]
We Don’t Trust Floating Point

- Doesn’t map perfectly to real numbers
- Can’t even represent 1/10 exactly
- Complex behavior of error and exceptions

But it’s what we’re stuck with

[1]
Floating-Point Arithmetic Is Not Associative

Let $\oplus$ be floating-point addition

- $0.1 \oplus (0.2 \oplus 0.3) = 0x1.3333333333334p-1$
- $(0.1 \oplus 0.2) \oplus 0.3 = 0x1.3333333333333p-1$
- Worse error when the magnitudes are different
Floating-Point Arithmetic Is Not Associative

Let $\oplus$ be floating-point addition

$0.1 \oplus (0.2 \oplus 0.3) = 0x1.33333333333334p-1$

$(0.1 \oplus 0.2) \oplus 0.3 = 0x1.3333333333333333p-1$

Worse error when the magnitudes are different

Does this bit matter?
Absolute vs. Relative Error

Let \( \hat{x} \) be an approximation for \( x \). Then relative error is

\[
\left| \frac{\hat{x} - x}{x} \right|
\]

and absolute error is

\[
|\hat{x} - x|
\]

- Think of absolute error as financial calculations; off by at most 1/10 cent (one mill)
- Think of relative error as significant digits
Let $\cdot$ be one of \{+$, -, \div, \times$\} and $\circ$ be its corresponding floating-point operation. Then

$$x \cdot y = (x \circ y)(1 + e) \text{ where } |e| \leq \epsilon.$$  \hfill (1)

- For double-precision $\epsilon = 2^{-53}$
- This holds only for $x \circ y \neq 0$ and normal (not subnormal)
Message Passing Interface (MPI)

- An API for communication between computers
- *de facto* standard for high-performance computing (HPC)
- Both “too high-level and too low-level” [8]
MPI Reduce

- Assume an array $A$ of size $n$
- Reduce $A$ to a single value
  - e.g. MPI\_SUM
- Distribute $A$ across MPI ranks (each $p_k$)
- Unspecified but usually deterministic reduction order on the same topology
How many ways are there to do this reduce?

- Depends on how we define acceptable reduction strategy
- We list four families
  1. Canonical Left-Associative (Canon)
  2. Fixed Order, Random Association (FORA)
  3. Random Order, Random Association (RORA)
  4. Random Order, Left-Associative (ROLA)
1. Canonical Left-Associative

- Left-associative
- Unambiguous: one reduction strategy
- No freedom to exploit parallelism

```c
double acc = 0.0;
for (i = 0; i < N; i++) {
    acc += A[i];
}
```
The MPI Standard is Flexible

- Operations are assumed to be associative and commutative.
- You may specify a custom operation where commutativity is fixed (but not associativity)
### Reduction Families Permitted by MPI

<table>
<thead>
<tr>
<th></th>
<th>Reduction Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fixed Order, Random Association (FORA)</td>
</tr>
<tr>
<td>3</td>
<td>Random Order, Random Association (RORA)</td>
</tr>
<tr>
<td>• Default if you call MPI_Reduce</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Random Order, Left Associative (ROLA)</td>
</tr>
<tr>
<td>• To compare with previous work [3]</td>
<td></td>
</tr>
</tbody>
</table>

- All of these have at least an exponential number of associations
- We generate these by shuffling an array, then generating random trees with Rémy’s Procedure [4, § 7.2]
Example Summation

With the commutative but nonassociative operator \( \oplus \),
\[ r_1 = r_2 \text{ but } r_2 \neq r_3. \]

\[
\begin{align*}
r_1 &= a \oplus (b \oplus c) \\
r_2 &= (c \oplus b) \oplus a \\
r_3 &= c \oplus (b \oplus a)
\end{align*}
\]
Absolute Error

Let \( \sum_{k=1}^{n} A_k \) be floating point sum, \( S_A \) be the true sum. Wilkinson back in ’63 proved summation error is bounded by

\[
\left| \sum_{k=1}^{n} A_k - S_A \right| \leq \epsilon (n - 1) \sum_{k=1}^{n} |A_k| + O(\epsilon^2).
\]
Left and Random Associativity (ROLA vs. RORA)

- Histogram of error
- 1000-digit float (MPFR) is true value
- ROLA is a biased sum
- worst RORA has smaller error than canonical

<table>
<thead>
<tr>
<th>Count</th>
<th>Zero</th>
<th>Canon</th>
<th>ROLA</th>
<th>RORA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLA</td>
<td>15000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RORA</td>
<td></td>
<td>10000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bound from (2): $4.44 \times 10^{-4}$
Nekbone

- Nekbone is a computational fluid dynamics proxy app
- We look at residual of conjugate gradient
- We use SimGrid [2] to try out 16 different allreduce algorithms
How can we apply high-level reasoning techniques about computer programs to low-level implementations? Specifically,

1. How can we write specifications of instruction set architectures (ISAs) that enable static analysis for program verification?

2. How can we formalize and quantify the error from floating-point arithmetic in high-performance numerical programs?
1. Introduction

2. Binary Analysis

3. A Statistical Analysis of Error in MPI Reduction Operations

4. Scalable Error Analysis for Floating-Point Program Optimization

5. Conclusion and Future Research Directions
The Challenges of Floating-Point

Suppose we want a safe floating-point divide? Easy, right?

```c
float unsafe(float x) {
    if (x==0.0)
        return 0.0;
    else
        return 1.0 / x;
}
```
The Challenges of Floating-Point

Suppose we want a safe floating-point divide? Easy, right?

```c
float unsafe(float x) {
    if (x==0.0)
        return 0.0;
    else
        return 1.0 / x;
}
```

Wrong
#include <math.h>

float reallysafe(float x) {
    // Cast to int without changing bits
    unsigned long c = *(unsigned long*)&x;
    if (isnan(x) || isinf(x) || (0x80000000 <= c && c <= 0x80200000) || (0x00000000 <= c && c <= 0x00200000))
        return 0.0;
    else
        return 1.0 / x;
}
Existing Static Error Analysis

- Tools like FPTaylor, Satire, Daisy
- Take as input a DSL describing a FP program and ranges of its inputs
- Output maximum possible error, found with global optimization
- No loops or conditionals
- Slow: \( \sim 1.5 \) hours for 500 FLOPs
- Most are sound
Why We Should Care About Soundness

- Underapproximating error may be worse than overapproximating
We previously saw $\epsilon$, the bound on relative error. For very small numbers, we must also define an absolute error

$$(x \odot y) = (x \cdot y)(1 + e) + d$$

where $|e| \leq \epsilon$, $|d| \leq \delta$.

e.g., $\delta = 2^{-1074}$ for double-precision
Motivation: Vector Normalization

Given a vector $x$, compute

$$q = \frac{x}{\|x\|_2}$$

Do this by multiplying each $x_i$ by $1/\sqrt{|x \cdot x|}$. 
Dot Products

Define

\[ \gamma_n = \frac{n\epsilon}{1 - n\epsilon}. \]

Unsound (existing bound)

\[ |\langle x, y \rangle - \text{flt}(x \cdot y)| \leq \gamma_n |x| \cdot |y| \]  \hfill (3)

Our improvement

\[ |\langle x, y \rangle - \text{flt}(x \cdot y)| \leq \gamma_n |x| \cdot |y| + n\delta(1 + \gamma_{n-1}). \]  \hfill (4)
My Key Insight

Combine global search for the hard parts and computed bounds for the majority of FLOPs

- FPTaylor on 500 FLOPs: 55,000 seconds
- FPTaylor + (4) on $10^9$ FLOPs: 10 seconds
- Speedup of $10^{11}$ - Not bad!
- Need to compare with empirical error
Reciprocal Square Root

Newton's Method, Initial Guess 1st Order Taylor

Input range and quality of initial guess have a large effect on convergence
1. Introduction

2. Binary Analysis

3. A Statistical Analysis of Error in MPI Reduction Operations

4. Scalable Error Analysis for Floating-Point Program Optimization

5. Conclusion and Future Research Directions
How can we apply high-level reasoning techniques about computer programs to low-level implementations? Specifically,

1. How can we write specifications of instruction set architectures (ISAs) that enable static analysis for program verification?

2. How can we formalize and quantify the error from floating-point arithmetic in high-performance numerical programs?
Future Research Directions

- Binary Analysis
- The Emerging Field of Formal Numerical Methods
  - Blend probabilistic and deterministic error analysis
- Precomputation, Once Again
Conclusion

▶ Verification of low-level programs is hard
▶ My techniques rely on detailed mathematical models and the speed of modern computers
▶ They help people write correct, fast code
  ● Quameleon: enables binary analysis on uncommon ISAs
  ● A statistical analysis of error for parallel reduction algorithms
  ● A sound analysis of error for optimized math kernels to quantify the performance-accuracy tradeoff
Conclusion

► Verification of low-level programs is hard
► My techniques rely on detailed mathematical models and the speed of modern computers
► They help people write correct, fast code
  ● Quameleon: enables binary analysis on uncommon ISAs
  ● A statistical analysis of error for parallel reduction algorithms
  ● A sound analysis of error for optimized math kernels to quantify the performance-accuracy tradeoff

https://sampollard.github.io/research
Thank you!
Motivation for Precomputation: *Quake III: Arena*

```c
float Q_rsqrt(float number) {
    long i;
    float x2, y;
    const float threehalves = 1.5F;
    x2 = number * 0.5F;
    y = number;
    i = *(long *) &y;
    i = 0x5f3759df - (i >> 1);
    y = *(float *) &i;
    return y;
}
```

“Magic” constant 0x5f3759df precomputed for efficiency [5]
Motivation for Precomputation: *Quake III: Arena*

```c
float Q_rsqrt(float number) {
    long i;
    float x2, y;
    const float threehalves = 1.5F;
    x2 = number * 0.5F;
    y = number;
    i = *(long *) &y;
    i = 0x5f3759df - (i >> 1);
    y = *(float *) &i;
    y = y * (threehalves - (x2*y*y));
    return y;
}
```

“Magic” constant 0x5f3759df precomputed for efficiency [5]

What does this do to a real number?
Fighting covid-19 using molecular dynamics simulations. 

Versatile, scalable, and accurate simulation of distributed applications and platforms.
*Journal of Parallel and Distributed Computing* 74, 10 (June 2014), 2899–2917.

[3] Chapp, D., Johnston, T., and Taufer, M.
On the need for reproducible numerical accuracy through intelligent runtime selection of reduction algorithms at the extreme scale.
In *IEEE International Conference on Cluster Computing* (Chicago, IL, USA, Sept. 2015), IEEE, pp. 166–175.

Addison-Wesley, Boston, MA, USA, 2006.
[5] Lomont, C.

Hulette, G. C., and Harrison, C. B.
Quameleon: A lifter and intermediate language for binary analysis.
In Workshop on Instruction Set Architecture Specification (Portland, OR, USA,

[7] Pollard, S. D., and Norris, B.
A statistical analysis of error in MPI reduction operations.
In Fourth International Workshop on Software Correctness for HPC Applications
[8] Snir, M.
Mpi is too high-level; mpi is too low-level.

An extra bit of analysis for clemency.
[7] and [6] part of this dissertation

A statistical analysis of error in MPI reduction operations.

Quameleon: A lifter and intermediate language for binary analysis.

A performance and recommendation system for parallel graph processing implementations: Work-in-progress.
Acceptance Rate: 43% (10/23).
Evaluation of an interference-free node allocation policy on fat-tree clusters. 
In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC ’18*, pages 26:1–26:13, Dallas, TX, USA, 
Acceptance rate: 24% (68/288).

A shared-memory algorithm for updating tree-based properties of large dynamic networks. 

A comparison of parallel graph processing implementations. 