

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

Samuel D. Pollard



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



Framing My Thesis

- ▶ 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.



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



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]

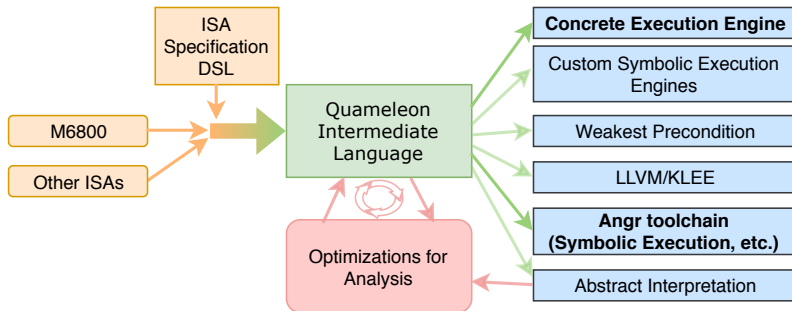


Motivation

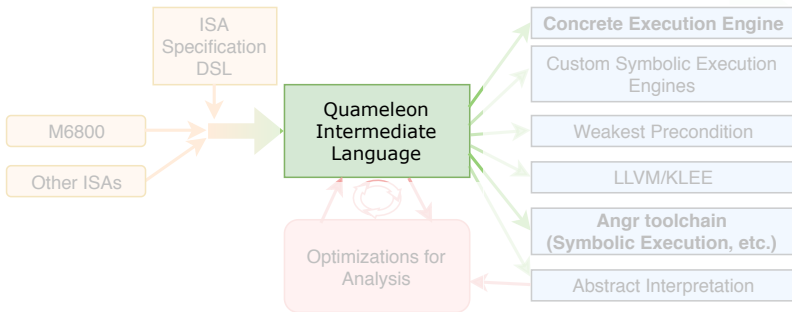
- ▶ Need to analyze binaries on old, obscure ISAs
 - ISAs not supported by existing tools
 - No machine-readable specification
 - Bad old days: No IEEE 754 floats, no 8-bit bytes
- ▶ Other tools gain lots of efficiency from expressive ISAs and feature-rich Intermediate Languages (ILs)
- ▶ We instead require an adaptable IL

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

Architectural Overview



Architectural Overview





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

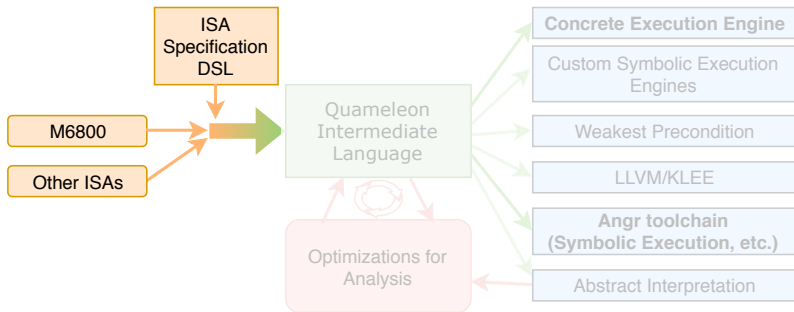


Quameleon Intermediate Language (QIL)

- ▶ 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]
```

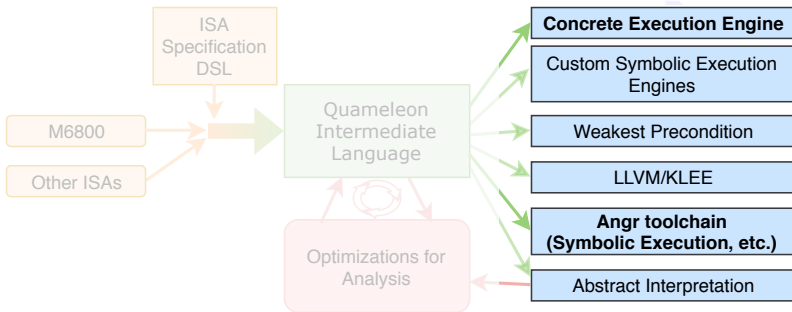
We want to match the manual precisely



...and Its Corresponding Semantics

```
AND r 1 -> do
  ra <- getRegVal r
  op <- loc8ToVal 1 -- Loc. of 8 bits in RAM
  rv <- andBit ra op
  z <- isZero rv
  writeReg r rv
  writeCC Zero z -- CC = Condition Code
  branch next
```

Back-ends

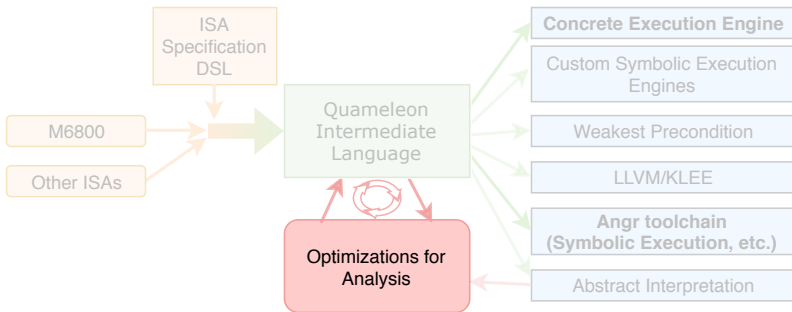




Current Back-ends

- 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





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

**Reduce
code
size**

**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)



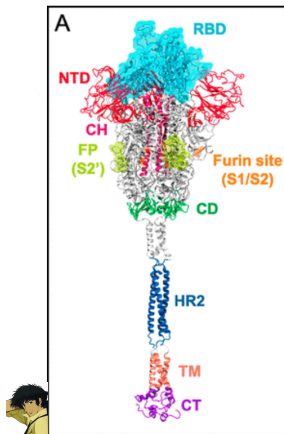
← Ariane V, the
\$500 million
overflow

✓ Done! You have submitted your taxes. Congratulations!



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

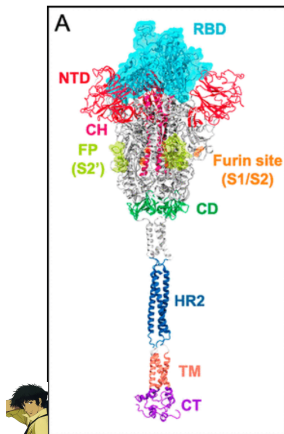


[1]

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
- ▶ $0.1 \oplus (0.2 \oplus 0.3) = 0x1.3333333333334p-1$
- ▶ $(0.1 \oplus 0.2) \oplus 0.3 = 0x1.3333333333333p-1$
- ▶ 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



Bound on Relative Error

- ▶ Let \cdot be one of $\{+, -, \div, \times\}$ and \odot be its corresponding floating-point operation. Then

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

- ▶ For double-precision $\epsilon = 2^{-53}$
- ▶ This holds only for $x \odot y \neq 0$ and normal (not subnormal)



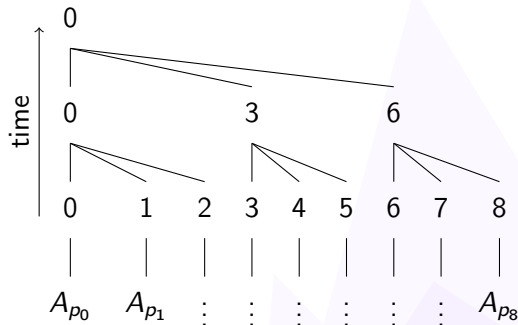
Message Passing Interface (MPI)

- ▶ 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

```
|| 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

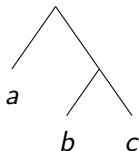
- 2 Fixed Order, Random Association (FORA)
- 3 Random Order, Random Association (RORA)
 - Default if you call `MPI_Reduce`
- 4 Random Order, Left Associative (ROLA)
 - To compare with previous work [3]
- ▶ All of these have at least an exponential number of *associations*
- ▶ We generate these by shuffling an array, then generating random trees with Rémy's Procedure [4, § 7.2]



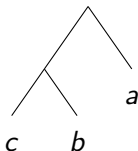
Example Summation

With the commutative but nonassociative operator \oplus ,
 $r_1 = r_2$ but $r_2 \neq r_3$.

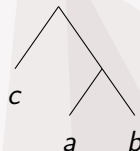
$$r_1 = a \oplus (b \oplus c)$$



$$r_2 = (c \oplus b) \oplus a$$



$$r_3 = c \oplus (b \oplus a)$$





Absolute Error

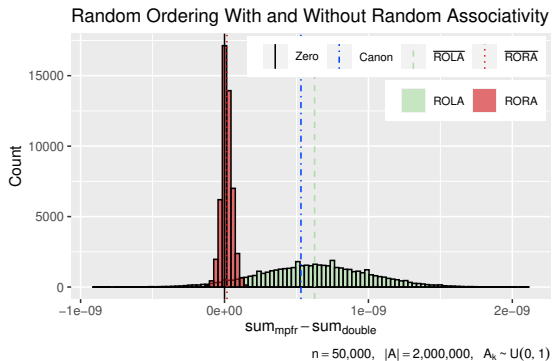
Let \sum^{\oplus} be floating point sum, S_A be the true sum.
Wilkinson back in '63 proved summation error is bounded by

$$\left| \sum_{k=1}^{\oplus n} A_k - S_A \right| \leq \epsilon(n-1) \sum_{k=1}^n |A_k| + O(\epsilon^2). \quad (2)$$



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

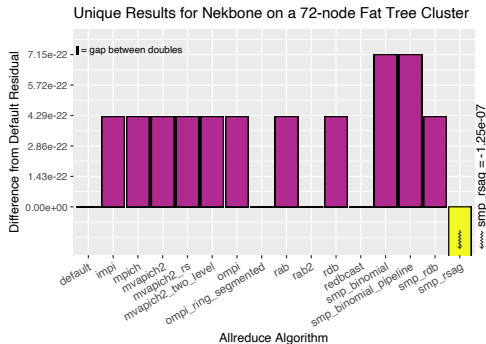


Bound from (2): 4.44×10^{-4}



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





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



The Challenges of Floating-Point

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

```
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?

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

wrong



Truly Safe FP Divide

```
#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: ~ 1.5 hours for 500 FLOPs
- ▶ Most are sound

Why We Should Care About Soundness



- Underapproximating error may be worse than overapproximating



Subnormal Numbers

We previously saw ϵ , 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)

$$|\langle x, y \rangle - \text{flt}(x \cdot y)| \leq \gamma_n |x| \cdot |y| \quad (3)$$

Our improvement

$$|\langle x, y \rangle - \text{flt}(x \cdot y)| \leq \gamma_n |x| \cdot |y| + n\delta(1 + \gamma_{n-1}), \quad (4)$$

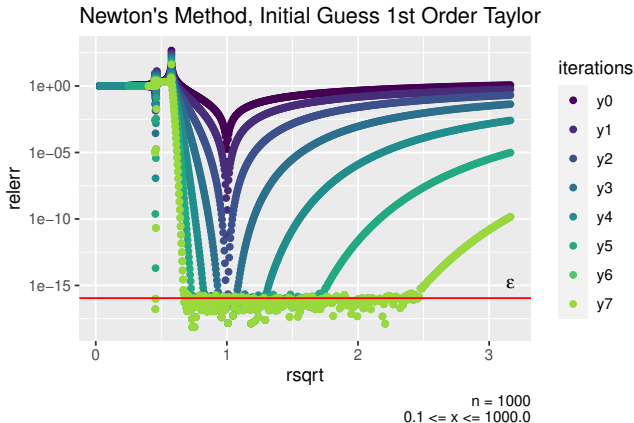


My Key Insight

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



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



Well... Did I Answer It?

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*

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

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



Motivation for Precomputation: *Quake III: Arena*

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

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

What does this
do to a real
number?

References I

- [1] Arantes, P. R., Saha, A., and Palermo, G.
Fighting covid-19 using molecular dynamics simulations.
ACS Central Science 6, 10 (2020), 1654–1656.
- [2] Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F.
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.
- [4] Knuth, D. E.
The Art of Computer Programming: Generating All Trees; History of Combinatorial Generation, vol. 4 Fascicle 4.
Addison-Wesley, Boston, MA, USA, 2006.

References II



- [5] Lomont, C.
Fast inverse square root, 2003.
Available at <http://lomont.org/papers/2003/InvSqrt.pdf>.
- [6] Pollard, S. D., Johnson-Freyd, P., Aytac, J., Duckworth, T., Carson, M. J., 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, Sept. 2019), SpISA '19, pp. 1–4.
- [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* (Nov. 2020), Correctness, IEEE, pp. 49–57.

References III

- [8] Snir, M.
Mpi is too high-level; mpi is too low-level.
MPI Symposium: 25 Years of MPI, Sept. 2017.
Available at https://www.mcs.anl.gov/mpi-symposium/slides/marc_snir_25yrsmpi.pdf.
- [9] Trail of Bits.
An extra bit of analysis for clemency.
Available at <https://blog.trailofbits.com/2017/07/30/an-extra-bit-of-analysis-for-clemency/>.

Publications I

[7] and [6] part of this dissertation

[10] Samuel D. Pollard and Boyana Norris.

A statistical analysis of error in MPI reduction operations.

In *Fourth International Workshop on Software Correctness for HPC Applications*, Correctness, pages 49–57. IEEE, November 2020.

[11] Samuel D. Pollard, Philip Johnson-Freyd, Jon Aytac, Tristan Duckworth, Michael J. Carson, Geoffrey C. Hulette, and Christopher B. Harrison.

Quameleon: A lifter and intermediate language for binary analysis.

In *Workshop on Instruction Set Architecture Specification*, SpISA '19, pages 1–4, Portland, OR, USA, September 2019.

[12] Samuel D. Pollard, Sudharshan Srinivasan, and Boyana Norris.

A performance and recommendation system for parallel graph processing implementations: Work-in-progress.

In *Companion of the 10th ACM/SPEC International Conference on Performance Engineering*, ICPE '19, pages 25–28, Mumbai, India, April 2019. ACM.
Acceptance Rate: 43% (10/23).

Publications II



- [13] Samuel D. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele.
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, November 2018. IEEE Press.
Acceptance rate: 24% (68/288).
- [14] Sriram Srinivasan, Samuel D. Pollard, Sajal K. Das, Boyana Norris, and Sanjukta Bhowmick.
A shared-memory algorithm for updating tree-based properties of large dynamic networks.
IEEE Transactions on Big Data, pages 1–15, September 2018.
- [15] Samuel D. Pollard and Boyana Norris.
A comparison of parallel graph processing implementations.
In IEEE International Conference on Cluster Computing, CLUSTER, pages 657–658, Honolulu, HI, USA, September 2017. IEEE Computer Society.