WHEN DOES A BIT MATTER? TECHNIQUES FOR VERIFYING THE CORRECTNESS OF ASSEMBLY LANGUAGES AND FLOATING-POINT PROGRAMS

by

SAMUEL DOUGLAS POLLARD

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2021

DISSERTATION APPROVAL PAGE

Student: Samuel Douglas Pollard

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

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Boyana Norris	Chair
Zena M. Ariola	Core Member
Hank Childs	Core Member
Benjamin Young	Institutional Representative

and

Andy Karduna	Interim Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded June 2021

DISSERTATION ABSTRACT

Samuel Douglas Pollard

Doctor of Philosophy

Department of Computer and Information Science

June 2021

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

This dissertation is about verifying the correctness of low-level computer programs. This is challenging because low-level programs by definition cannot use many useful abstractions of computer science. Features of high-level languages such as type systems or abstraction over binary representation of data provide rich information about the purpose of a computer program, which verification techniques or programmers can use as evidence of correctness.

Sometimes, however, using these abstractions is impossible. For example, compilers use binary and assembly-level representations and sometimes performance constraints demand hand-written assembly. With numerical programs, floating-point arithmetic only approximates real arithmetic. Floating-point issues are compounded by parallel computing, where a large space of solutions are acceptable.

To reconcile this lack of abstraction, we apply high-level reasoning techniques to help verify correctness on three different low-level programming models computer scientists use. First, we implement a binary analysis tool to formalize assembly languages and instruction set architectures to facilitate verification of binaries. We then look at floating-point arithmetic as it applies to parallel reductions. This expands the notion of reductions to one which permits the many different solutions allowed by the Message Passing Interface (MPI) standard. Last, we refine floating-point error analysis of numerical kernels to quantify the tradeoff between accuracy and performance. These enhancements beyond traditional definitions of correctness help programmers understand when, and precisely how, a computer program's behavior is correct.

This dissertation includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Samuel Douglas Pollard

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Western Washington University, Bellingham, WA, USA

DEGREES AWARDED:

Doctor of Philosophy in computer science, 2021, University of Oregon
Master of Science in computer science, 2016, Western Washington University
Bachelor of Science in mathematics, 2014, Western Washington University

AREAS OF SPECIAL INTEREST:

High Performance Computing
Computer Arithmetic
Formal Methods

PROFESSIONAL EXPERIENCE:

Formal Methods Summer R&D S&E Intern, Sandia National Laboratories, Livermore, CA, Summers of 2018, 2019, 2020
Formal Methods Year-Round R&D S&E Intern, Sandia National Laboratories, Livermore, CA, Sep. 2018–Jun. 2021
Computation Student Intern, Lawrence Livermore National Laboratory, Livermore, CA, Summer 2017
Graduate Employee, University of Oregon. Sep. 2016–Jun. 2021

GRANTS, AWARDS AND HONORS:

University of Oregon General University Scholarship, 2020
University of Oregon General University Scholarship, 2019
ACM/SPEC 2019 International Conference on Performance Engineering Travel Grant
IEEE Cluster 2017 Travel Grant
Erwin & Gertrude Juilfs Scholarship in Computer and Information Science, 2017

PUBLICATIONS:

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.

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.

Pollard, S. D., Srinivasan, S., and Norris, B. 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* (Mumbai, India, Apr. 2019), ICPE '19, ACM, pp. 25–28.

Pollard, S. D., Jain, N., Herbein, S., and Bhatele, A. 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* (Dallas, TX, USA, Nov. 2018), SC '18, IEEE Press, pp. 26:1–26:13.

Srinivasan, S., Pollard, S. D., Das, S. K., Norris, B., and Bhowmick, S. A shared-memory algorithm for updating tree-based properties of large dynamic networks. *IEEE Transactions on Big Data* (Sept. 2018), 1–15.

Pollard, S. D., and Norris, B. A comparison of parallel graph processing implementations. In *IEEE International Conference on Cluster Computing* (Honolulu, HI, USA, Sept. 2017), CLUSTER, IEEE Computer Society, pp. 657–658.

ACKNOWLEDGEMENTS

First I give thanks to all the organizations and people who generously supported my Ph.D.: Lawrence Livermore National Laboratory under the auspices of the Department of Energy under Contract DE-AC52-07NA27344 (LLNL-CONF-745526), the Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357, The National Science Foundation Computing and Communications Foundations Award #1725585, and Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525, Sand No. SAND2021-6801 T.

Next, I thank my mentors and professors across my university career. First and foremost, my advisor Boyana Norris who supported me through all the various topics across which my studies meandered. Thanks to Zena Ariola for her guidance and for connecting me with the most relevant researchers and papers. And also thanks to Perry Fizzano and Tjalling Ypma at Western Washington University for their mentorship, which helped inspire me to pursue a Ph.D.

I am grateful to my colleagues, co-authors, and mentors at Sandia Labs, especially Philip Johnson-Freyd and Geoff Hulette for first introducing me to formal methods and teaching me the most important concepts in the field.

I also would like to thank everyone who provided input on this dissertation. First, thanks to Augustin Degomme and the SimGrid Team for their help with SimGrid, Dylan Chapp for explaining his research, and Jackson Mayo for his insightful comments on Chapter IV. I also give thanks to Zach Sullivan for his detailed and helpful critiques of the final dissertation document.

Last, I thank my parents for their unwavering support and their dedication to my own independence, ever since the third grade when they said they couldn't (or wouldn't?) help me with math homework anymore.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

LIST OF TABLES

Table Page

LIST OF SOURCE CODE LISTINGS

CHAPTER I

INTRODUCTION

Something weird might just be something familiar viewed from a different angle, and that's not scary. Right?

Hannah K. Nyström

A fundamental concept in computer science is that of abstraction. Over the past sixty years, two abstractions have become especially successful: instruction set architectures (ISAs) and floating-point (FP) arithmetic. The former separates the design of a computer's processor with its interface, while the latter approximates real arithmetic. One cause—and symptom—of their success is their transparency; these abstractions are so heavily used that it is easy for programmers to forget they exist. However, subtle and sometimes disastrous bugs can arise from misunderstandings of these abstractions. One reason bugs arise is the disconnect between informal specification and actual behavior. For example, errors relating to CPUs may be documentation bugs which go unnoticed for years [5] or cost hundreds of millions of dollars [68]. A much more serious bug regarding the computer representation of 1/10 in a missile guidance system resulted in dozens of deaths [24].

This thesis is about translating the kinds of high-level, informal specifications of ISAs and FP arithmetic into a specification which can be checked by a computer. The goal is to detect and prevent bugs which arise from the subtle differences between the high-level specifications and the actual behavior of computer programs. We begin with ISAs, where "correct" means every single bit of a processor's state is specified. We then extend our bit-level analysis to parallel FP arithmetic, where many different solutions must be acceptable.

Throughout this dissertation, we use the term *formal* often. This term has many meanings within formal methods, but across the field the definition is stronger than how formal is used in English. Loosely speaking, for a statement to be *formal* it must have some mathematical rigor and be expressed in a way which can be checked mechanically. For example, a tautology expressed in symbolic logic is formal. Under this strict definition, there are rarely formal specifications of ISAs or even mission-critical FP programs (that is, programs whose main outputs are floating-point numbers). For ISAs the informal specification is typically a reference manual written in English. This is where the technical definition is sometimes at odds with the vernacular; something as stodgy as an ISA reference manual or Ph.D. dissertation may be referred to as "formal" writing, but it is informal under our definition. A reference manual is informal because there is no mechanical way to ensure the statements are self-consistent or unambiguous since they are written in a natural language.

Similarly, specifications of numerical programs typically relate the program's behavior to real arithmetic and, unfortunately, the mapping between FP and real arithmetic is not exact. When considering parallel computations, FP arithmetic is even more complex because many different solutions must be acceptable to achieve the speedup of parallel execution.

Two more terms key to understanding this dissertation are *high-level* and *low-level*. In general, high-level statements are not concerned with the underlying implementation of a program, but "high" and "low" depend on context. With ISAs, a high-level statement may be described in a high-level programming language (for example, assertions written in C).

For general computer programs, *high-level* reasoning means deducing some property *about* the computer program. To be a little more specific, consider a field of automated reasoning called automatic differentiation (AD). Using sophisticated algorithms, AD can describe the derivative of a function written in a computer code. With FP arithmetic, high-level statements describe real numbers. In contrast, low-level properties of a program must take into account implementation details such as the endianness of memory or the details of the FP standard used, most commonly the IEEE 754 standard. To illustrate one difficulty with reconciling high-level and low-level abstractions, we note that $(a + b) + c$ is true for real numbers, but this is not in general true for FP numbers. This means a programmer or term rewriting system cannot use a high-level property (associativity) because of the limitation of the underlying representation (IEEE 754).

Now that we have framed the challenge of reconciling high-level and low-level abstractions, we are ready to pose the question this dissertation seeks to answer: *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?*

This dissertation develops an answer to this question through four more chapters, structured as follows. In Chapter II, we give an introduction to formal methods as a field and some of the main strategies verification engineers use to reason about program correctness, such as deductive program verification, abstract interpretation, satisfiability modulo theories (SMT) solvers, and proof assistants. This chapter also contains an introduction to floating-point arithmetic, its binary representation, and tools for FP error analysis. We then move on to answering the first part of this dissertation question.

ISAs separate the implementation details of a microprocessor from its interface—effectively providing a barrier between computer science and electrical engineering. The way these ISAs are described is typically through manuals which describe the ISA's semantics informally via diagrams and natural language (e.g., English) descriptions. However, programmers must translate these informal semantics into formal semantics to be used in computer programs such as compilers. Because these ISAs are at a low level of abstraction, bugs arising from ISA specifications or their translation into a computer program can be subtle and difficult to debug. For an ISA to be formally verified, its semantics must be specified in a way that is machine-checkable. This has been done before, but only for new and open-source architectures [5]. The reality of the world is some of the most important computer programs that humanity uses *only* run on old and obscure architectures which are not formally specified.

To address these issues, Chapter III describes a binary analysis tool, called *Quameleon*, which provides an easier way to make assembly language specifications machine-readable, using type systems to prevent large classes of errors. Furthermore, careful design of Quameleon's intermediate language allows analysis of binaries independent of ISA or computer architecture. We show feasibility by exhaustively exploring a program's possible execution paths using symbolic execution. This answers the first part of this dissertation's question because Quameleon allows translation of high-level concepts into a format that can be used to check properties of binaries and of the specification itself. High level in this context means those properties described in an ISA reference manual or assertions about programs written in languages like C. This chapter includes previously published co-authored material by Samuel D. Pollard, Philip Johnson-Freyd, Jon Aytac, Tristan Duckworth, Michael J. Carson, Geoffrey C. Hulette, and Christopher B. Harrison [151].

In the field of numerical analysis, sophisticated theories exist to precisely describe error bounds on algorithms like Newton's method. More generally, iterative methods have a notion of *numerical stability* expressed through properties of the system such as a matrix's condition number. These techniques work in some cases when applied to FP arithmetic, but in general FP numbers do not have many of the useful properties of real numbers such as associativity, and so traditional numerical analysis falls short.

In Chapter IV, we carry out a careful statistical analysis of FP arithmetic as it applies to parallel reductions to give more realistic confidence intervals. This refines analysis of parallel reductions by sampling from a more realistic space of different solutions: those allowed by the Message Passing Interface (MPI) standard. This chapter begins to answer the second part of this dissertation's question. Specifically, the MPI standard states only *that* different answers are permitted, not *what* they may look like. This provides a clearer picture of actual error for MPI reductions. This chapter includes previously published co-authored material by Samuel D. Pollard and Boyana Norris [152].

As mentioned before, most numerical algorithms require some notion of *numerical stability*. Informally, this is the property that small roundoff or sampling errors in an algorithm do not accumulate to large magnitudes in the output. For example, FP subtraction can be unstable because $1 - (1 + x) \neq x$ for sufficiently small x. Numerical stability depends on the problem domain and particular algorithm and so in general it is a vague term. However, stability is highly important to numerical analysts and numerical library developers. Because numerical algorithms usually take up the vast majority of the execution time for high-performance computing (HPC) applications, HPC developers must balance numerical stability and performance for highly-optimized computational kernels (which are small but important algorithms) such as the linear algebra subroutines of BLAS [23].

In Chapter V, we refine and further quantify the notion of numerical stability by giving a more principled look at error across various input spaces. We accomplish this by first visualizing error for numbers very close to 0 (subnormal numbers) which are often overlooked in FP error analysis. We next aim for scalable error analysis on vectors of FP numbers by providing improved error bounds for subnormal arithmetic on inner products. Our work is not the first to analyze error propagation rigorously. However, our work achieves scalability of FP error analysis across all FP values (normal and subnormal). Then, this analysis is combined with a search to find the global maximum FP error from a tool called FPTaylor [178] to analyze the error of more complex FP kernels. This further answers the second part of this dissertation's question by providing numerical analysts with precise bounds on numerical error and expected performance characteristics. We include information to aid in the reproducibility of the work in this dissertation in Appendices A and B.

In conclusion, this dissertation provides programmers the ability to perform static analysis on families of binaries in ways that were not previously possible. We accomplish this by investigating what high-level statements about computer programs actually mean when translated to their binary implementations. We use these techniques in two main application domains: ISAs and FP arithmetic. The former is already a well-established area of study, both related to hardware and compiler design. The latter, namely the union of formal methods techniques and numerical analysis, is becoming a field in its own right. Chapter VI provides some unifying remarks and potential research directions in this exciting new area.

CHAPTER II
BACKGROUND

2.1 Verification of Computation

Verification of the behavior of a computation predates electronic computers, but a reasonable starting point is 1940 with Alonzo Church's invention of simply typed lambda calculus (STLC) [40]. At first, STLC was invented to prevent paradoxes like Curry's paradox. Further study of STLC inspired computer scientists and mathematicians to develop increasingly powerful type theory. Programming languages with these features not only help avoid logical paradoxes and prevent common programming errors but also have deep connections with formal proofs. By the Curry-Howard isomorphism, proof systems in intuitionistic logic—a logic in which all proofs are constructive and thus computable—correspond exactly to typed models of computation [179].

Alongside the exponential increase in the complexity of computers, two interesting phenomena occurred. The first is the increasing abstraction computer scientists developed to manage this complexity. The second is the development of sophisticated type systems, logics, and proof systems such as the Calculus of Inductive Constructions (CIC) [49], separation logic [156], and sequent calculus [197]. These allow expressive logical specifications of program behavior and machine-checked proofs that implementations match these specifications.

Despite this move towards the highly abstract and expressive, microprocessors still operate on low-level, imperative machine languages. Ultimately, there is still a need for computer programs that are written at the lowest level of abstraction: bootloaders, performance-critical code, and compiler optimizations are performed on languages that interact directly with microprocessor hardware. Additionally, the increasing heterogeneity of hardware with the advent of graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and other accelerators brings with it the need for more low-level code. This hardware often has vastly different behavior and performance characteristics which makes correct and well-performing code difficult to write—and correct, well-performing, portable code nearly impossible.

This chapter presents the most common formal methods techniques and how they are used to verify low-level programs. We emphasize that verifying any program hinges on a description of what we wish to verify—the *specification*. In practice, these specifications range from informal English descriptions (for example, "This program shall not crash on any input") to more official English specifications (such as Internet Engineering Task Force Requests for Comments (RFC) or ISO standards) to machine-readable and machine-checkable specifications.

Ultimately, all of formal methods fall into two domains. These are so central to the field we distinguish them typographically:

Definition 2.1. Formal specification*: writing precise, unambiguous descriptions of what we want a program to do.*

Definition 2.2. Formal verification*: proving the program is correct with respect to this specification.*

Verifying any property of a program requires both domains. We can think of specification as the "front-end" of formal methods while verification is the "back-end."

Another bifurcation of formal methods is *model checking* versus *correct by construction programming* (CBCP). With model checking, the desired program's behavior is described using a specification language such as TLA+[119], or $F^{\star}$ [183] and a model is generated from that specification. The model is then checked (via exhaustive state space search, for example) by a computer program to ensure it meets the specification. Correct by construction programming instead begins with the user developing in a language with features that allow the generation and dispatch of proofs of correctness. Examples include Coq [186], HOL/Isabelle [147], and Lean [61]. We typically get a higher level of assurance with CBCP at the cost of rewriting an application in a different language. In practice, approaches typically blend model checking and CBCP.

To understand how one may go about verifying computation, we first introduce some logical notation and concepts used throughout formal methods in Section 2.2. We then describe some static analysis techniques such as abstract interpretation in Section 2.3. We further describe a workflow for formal methods and some tools and languages for model checking and CBCP in Section 2.4. The main portion of this chapter describes how these verification techniques are applied to intermediate representations (IRs), assembly language programs, floating-point arithmetic, and parallelism as it relates to numerical computations (Section 2.5–2.8). We conclude this chapter with some references to formal methods' applications and adoption (Section 2.9).

2.2 Logical Notation

We provide a brief introduction to logic chiefly to establish a consistent notation. An excellent, more comprehensive introduction is given in the Stanford Encyclopedia of Philosophy [173]. A *predicate* or *formula* is a function mapping from some *domain* into the set

{True, False}. A domain is all mathematical objects under consideration. An example of a domain is the set of all pairs of integers ($\mathbb{Z} \times \mathbb{Z}$) which an accompanying predicate may be $P(x, y) = x + y > 0$. Here, x and y are *variables* and may only be assigned values from their appropriate sets.

Now that we have defined predicates, the next natural step is reasoning about how to combine and modify them. A branch of logic that accomplishes this is *propositional logic*. Predicates in propositional logic may only contain variables and the following logical operators. These are also called *connectives* because they connect one or two objects. The propositional logic connectives are:

- and ($\wedge$);
- or ($\vee$);
- exclusive or (xor);
- implies ($\implies$);
- if and only if ($\iff$);
- not ($\neg$).

First-order logic contains all of propositional logic with the addition of quantifiers:

- existential ($\exists$);
- universal ($\forall$).

In first-order logic, these quantifiers may only be applied to objects in the domain (as opposed to applying to other quantifiers). For example, $\forall x, y \in \mathbb{R}, x + y = y + x$ is a valid first-order predicate. The symbol $\exists$ is read as "there exists", $\forall$ as "for all," and often a period is used to separate the statements of an existential quantifier—that period is read as "such that."

Second-order logic allows these quantifiers to be applied not just to objects in the domain but to properties (that is, formulas consisting only of logical connectives, quantifiers, and objects). For example, suppose our domain is the set of pairs of real numbers ($\mathbb{R} \times \mathbb{R}$) and we wish to say there is some property that holds regardless of how you pass in the arguments.

$$\exists P(x, y).\ \forall x, y\ P(x, y) \iff P(y, x) \tag{2.1}$$

where the first $\exists$ is the second-order quantifier. One example of a property which can be used to prove (2.1) is the statement "you can always add two real numbers to get a third." Put symbolically,

$$P(x, y) := \forall x, y \; \exists z. \; x + y = z. \tag{2.2}$$

We know (2.2) to be true by commutativity of $\mathbb{R}$, thus we can use it to prove (2.1).

A non-example is division; 0/1 is defined but 1/0 is not. Namely,

$$Q(x, y) := \forall x, y \; \exists z. \; x/y = z$$

is false. Higher-order logic includes second-order logic, third-order logic, fourth, etc. Third-order logic, for example, would allow quantifiers to be applied to properties about properties.

When comparing propositional, first-order, and higher-order logic, expressiveness increases while the ability to reason about the logic decreases. For example, specifying the rules of chess using first-order logic requires one page of definitions, whereas using propositional logic requires 100,000 pages [167]. Conversely, we can decide the satisfiability of any propositional logic formula, but in general, this is impossible for first-order logic. Now that we have given a brief introduction to logic, we have the background to introduce static analysis, the main formal method we use in this dissertation to verify low-level programs.

2.3 Static Analysis

Static analysis is a broad term for analyzing programs without executing them. Static analysis is important to this dissertation since it is the main formal method we utilize throughout. Chapter III uses symbolic execution to prove properties of binaries and uses compile-time checking of assembly languages to detect bugs. Chapter IV provides expected error intervals for parallel floating-point subroutines. This allows users to estimate errors from parallel summations based on the statistical distribution of inputs. Lastly, Chapter V builds on existing floating-point static analysis tools to provide error bounds for common floating-point subroutines which form the basis of numerical algorithms.

2.3.1 Abstract Interpretation. Abstract interpretation was invented by Radhia and Patrick Cousot [50] in 1977. Abstract interpretation has a theoretical framework based on Galois connections but intuitively we can think of abstract interpretation as "sound static analysis." Sound means that any property discovered by the static analyzer will also hold when executing

Table 1. Abstract valuation for division. The column is the numerator and the row is the denominator. We bold the column used in Listings 2.1, 2.2, and 2.3.

/	0	**nnz**	NaN	$\pm\infty$	flt
0	NaN	$\pm\infty$	NaN	$\pm\infty$	flt
nnz	flt	**flt**[1]	NaN	$\pm\infty$	flt
NaN	NaN	**NaN**	NaN	NaN	NaN
$\pm\infty$	0	**0**	NaN	NaN	flt
flt	flt	**flt**	NaN	flt	flt

[1] For a numerator too close to 0, underflow to 0 can occur. For a denominator too close to 0, overflow to $\pm\infty$ can occur.

the program—no false positives. An accessible introduction to abstract interpretation was published in 2020 [158].

The key to abstract interpretation is choosing an *abstract domain* over which to analyze. An example of an abstract domain over the floating-point (FP) numbers would be to consider every FP value as either zero, nonzero, infinite, or Not-a-Number (NaN). Thus the abstract domain is the set

$$\mathcal{A} = \{0, \text{nnz}, \pm\infty, \text{NaN}, \text{flt}\}. \tag{2.3}$$

We require the last element "flt" which represents *all* FP numbers, because for some operations we cannot know for certain what the value will be. We are careful to distinguish here between real-valued division and FP division. Real division is denoted / and FP division $\oslash$. For example, $\mathtt{f}(\mathtt{x}) = 1.0 \oslash \mathtt{x}$ will not return finite values for all $\mathtt{x}$.

With this abstract domain, we could check if an FP operation "behaves nicely," depending on our definition of nice. One example of "nice" might be a division operation which always returns a finite value. To use (2.3) as our abstract domain, we must calculate the abstract valuation for the operations we care about. Here we choose only division and show the valuation in Table 1.

To see the use of the abstract domain $\mathcal{A}$, we describe three different implementations of division in Listings 2.1, 2.2, and 2.3. Each is attempting to compute $\mathtt{f}(\mathtt{x}) = 1.0 \oslash \mathtt{x}$ with the additional goal that `f` returns either $1.0 \oslash \mathtt{x}$ if finite, or 0.0 in any other case. We notice only the "nnz" column is necessary since the numerator is always 1.0. Therefore, we must consider several cases to ensure `f` returns a finite value. The first, Listing 2.1, only considers $\mathtt{x} = 0.0$. An abstract interpreter could explore both branches of the conditional and notice in the `else` branch it cannot

Listing 2.1 Unsafe floating-point division.

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

Listing 2.2 Floating-point division checking most—but not all—cases where the result is not finite.

```c
#include <math.h>
float mostlysafe(float x) {
    if (x==0.0 || isnan(x) || isinf(x))
        return 0.0;
    else
        return 1.0 / x;
}
```

narrow down the scope of x sufficiently, thus will state that the function `unsafe` will return some flt.

For Listing 2.2, we consider the three edge cases for x: 0, NaN, $\pm\infty$. These all behave correctly, but for some nnz we still cannot ensure that $1.0 \oslash x$ is finite because the reciprocal of sufficiently small floating-point numbers can overflow to $\pm\infty$. Therefore, `mostlysafe` will also return flt.

This is somewhat of an unsatisfying result. We present this example because, despite its relative simplicity, we still require a more complex abstract domain to prove correctness. We discuss the subtleties of floating-point arithmetic in further detail in Section 2.6. But fear not, to get some closure we include Listing 2.3 which always returns a finite value and returns the floating-point approximation of $1/x$ for exactly the set of x which do not overflow. We found these tight bounds through an exhaustive search of the input space.

There is interest in using abstract interpretation over floating-point operations for parallel reproducible linear algebra subroutines (ReproBLAS) [64] in applications such as medical diagnosis and legal cases; we go into more detail in such applications in Chapter V.

2.3.2 Symbolic Execution. Symbolic execution is based on the idea that a program's variables can represent logical formulas rather than concrete values [30, 44]. Intuitively, symbolic execution transforms the types of all variables from their concrete types (such as integers or

Listing 2.3 Floating-point division guaranted to always return a finite value.

```c
#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;
}
```

strings) into symbolic expressions representing how values of that concrete type are transformed by a given program or function. Symbolic execution is a form of static analysis which generates formulas in propositional logic to be dispatched to Satisfiability Modulo Theories (SMT) solvers. These are described more in Section 2.3.3.

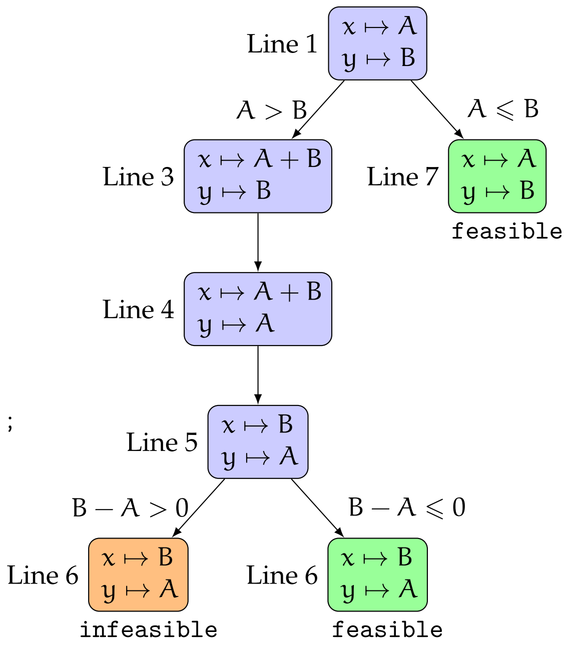
Symbolic execution is a popular technique with dozens of packages implementing techniques ranging from pure symbolic execution to a mix of concrete and symbolic (concolic) execution to generate test cases for a given program. A survey of symbolic execution techniques is given by Baldoni et al. [11]. Some popular symbolic execution engines include angr [175], primarily designed for security, and KLEE [36], primarily designed for finding bugs.

Figure 1 shows a conceptual example of how symbolic execution proceeds. We begin by mapping x and y to symbolic values A and B, which represent any possible input. At the `if` statement at line 2, there is a branch in the symbolic executor that splits the execution into two possibilities, which it must then explore completely. The right branch is easy because we are done with the program and no assertions have failed. The left branch is more interesting. Consider line 4, the statement `y = x - y;`. Symbolically, our previous state is

$$x \mapsto A + B,\ y \mapsto B$$

so subtracting $x - y$ is symbolically equivalent to $A + B - B = A$. This sort of transformation is the core of symbolic execution. Proceeding down the tree and likewise the program, we reach the single assertion on line 6. Substituting the symbolic values in for x and y we get an infeasible (unreachable) path for $B - A > 0$ and feasible (reachable) otherwise. This allows us to verify the

```
1 void f(int x, int y) {
2   if (x > y) {
3     x = x + y;
4     y = x - y;
5     x = x - y;
6     assert(x - y > 0);
7   }
8   printf("(%d,%d)\n",x,y);
9 }
```



Line 1
$x \mapsto A$ $y \mapsto B$
$A > B$
$A \leqslant B$
Line 3
$x \mapsto A + B$ $y \mapsto B$
Line 7
$x \mapsto A$ $y \mapsto B$
feasible
Line 4
$x \mapsto A + B$ $y \mapsto A$
Line 5
$x \mapsto B$ $y \mapsto A$
$B - A > 0$
$B - A \leqslant 0$
Line 6
$x \mapsto B$ $y \mapsto A$
infeasible
Line 6
$x \mapsto B$ $y \mapsto A$
feasible

Figure 1. An example illustrating symbolic execution from Torlak [192]. Line numbers in the right graph indicate the state of the program *after* the listed line. Symbolic execution can determine there are no values of x and y that make the assertion fail.

assertion will never fail because the only state in which it does fail is unreachable since $A > B$ being true at the higher branch implies $B - A > 0$ is false at the lower branch.

One limitation of symbolic execution is the *state space explosion* problem. This arises from the fact that symbolic execution explores all potential execution paths. This is simple in Figure 1 but for programs with unbounded loops, it could be impossible to completely explore all paths. Even for relatively simple codes, full symbolic execution may be intractable. To mitigate this, researchers may maximize code coverage without exploring all paths or transforming programs to reduce the number of branches.

2.3.3 SMT: Satisfiability Modulo Theories. SMT solvers are to formal methods as matrix multiplication is to scientific computing: many problems can be expressed with respect to these algorithms, much work has gone into optimizing them, and they are often the most computationally expensive part of a larger algorithm.

Before we get too far ahead of ourselves, we will break down SMT. *Satisfiability* relates to the fact that the output of an SMT solver is essentially “yes” or “no” with the caveat that if “yes” the user may want to know *how* to satisfy the formulas. *Modulo Theories* means the question of satisfiability is identical but different theories can be substituted in terms. For example, typical SMT solvers have theories for integers, lists, arrays, and even floating-point numbers.

For example, if our SMT solver understands the theory of integer arithmetic then

$$x + 2y = 20 \wedge x - y = 2$$

is satisfiable because both equations are true for $x = 8$ and $y = 6$. An unsatisfiable example is

$$x > 0 \wedge y > 0 \wedge x + y < 0.$$

A good overview of SMT is given by Barrett [17].

Once a problem is reduced to a question of satisfiability, the problem can be expressed in a standardized API called SMTLIB [16] which is then passed into an SMT solver to compute the satisfiability of a given formula. Some popular SMT Solvers include CVC4 [15], Z3 [62], Alt-Ergo [115], and OpenSMT [34].

Abstract interpretation, symbolic execution, and SMT solvers are the building blocks of formal methods, however there are many more formal methods. Some other influential examples of strategies for verification are Hoare logic [66], and separation logic [156]. Proof assistants such as Coq [187] and HOL/Isabelle [147] provide powerful type theories and logical systems which can be used to verify program correctness.

2.4 Formal Methods in Practice

Given that a user wants to verify a piece of software, s/he must make a few design decisions. These begin intentionally vague.

1. To what degree of confidence must the software be guaranteed?
2. What tools can be used to accomplish #1?
3. How much time can a human spend on #2? How much time can a computer spend on #2?

Each has important considerations we address in the following subsections, respectively.

2.4.1 Degrees of Confidence. Given the task of verification, at what point do we become confident enough that a piece of software is correct? Donald Knuth's famous quote hints at this challenge: "Beware of bugs in the above code; I have only proved it correct, not tried it." Verily, bugs in software are ubiquitous. Additionally, verification is only as good as the specification. One example is a security vulnerability found in the WPA2 WiFi standard which had previously been proven secure [195]. The exploit did not violate the verified safety properties but instead exploited a temporal property, of which the WPA2 specification was ambiguous. While temporal properties are beyond the scope of this dissertation, we must always remember a bug in any part of the software toolchain, from handwritten specifications all the way down to hardware implementations, can be the cause for failure. For example, in 1962 NASA's Mariner I rocket crashed because of a mistake in the handwritten formula for a radar timing system [146]. Even worse, the accumulation of errors from representing 0.1 seconds in binary using a fixed-point format resulted in the deaths of 28 soldiers because of a timing error of a Patriot missile [24].

Furthermore, one must also know if one can trust the verifier; a bug in the verification software could cause false positives or false negatives. The approach to solving this problem taken by some proof assistants is to create a small, well-understood kernel upon which all else is built. The trust is further strengthened by proving existing theorems using this kernel, proving the kernel's correctness by hand, and checking the results on many computer systems. Having a simple, trusted kernel is known as the *de Bruijn* criterion. Put another way, we write a proof (which by the Curry-Howard correspondence is equivalent to the evaluation of a function), but we must believe the program which performs checks the proof is correct as well. The simpler this evaluator is, the easier it is to verify its correctness.

Epistemology aside, a more practical approach would be to determine what sort of properties you may want to prove about a system. For example, consider Listing 2.4 which attempts to find solutions to $a^3 + b^3 = c^3$.[1]

Now, there are several questions we may want to ask about this program. Some are basic safety guarantees. For example, will the function `ipow` correctly handle all integers? In this case, no because $n < 0$ will most likely cause a stack overflow. The next question we may ask is will the program ever pass a negative value into `ipow`? We say yes because eventually the integers will overflow into negative values and cause an error. These sorts of questions can be answered with symbolic execution then dispatched to SMT solvers.

A more interesting question is will this program test every valid combination of $a, b,$ and c? Proving this requires some loop invariants which requires annotating the source

[1]We use $n = 3$ to keep the algorithm simple, but it is important to note a proof for $n = 3$ has been known for hundreds of years.

Listing 2.4 A naïve attempt to find a counterexample to Fermat's Last Theorem for $n = 3$.

```
int ipow(int x, int n) {
  if (n==0) return 1;
  return x * ipow(x,n-1);
}
int main() {
  int a,b,c,n;
  n = 3; c = 0;
  while (1) {
    c++;
    for (a = 1; a < c; a++) {
      for (b = 1; b < c; b++) {
        if (ipow(a,n)+ipow(b,n)==ipow(c,n))
          return 0;
      }
    }
  }
}
```

code, but otherwise manageable. Once we've done this, we've proven partial correctness of this code. That is, if the program terminates, then it gives us the right answer.

The final question we may ask is: Does this program terminate? In order to answer this question for all $n > 2$, we need a proof of Fermat's Last Theorem, a feat which was left unanswered until Wiles' proof in 1995 [201]. Thus, we see the property we wish to prove about a program ranges from trivial to almost impossible.

2.4.2 Methods for. . . Formal Methods. When selecting a strategy for verification, one must know what sort of properties may be proven using that strategy. In his 2016 Milner Award Lecture, Xavier Leroy provides an illustrative image describing the tradeoff between interactive and automatic verification [126]. We render this image in Figure 2. We provide an overview in Table 2 and follow up with a more detailed description of each.

Many formal methods concepts rely on these automated reasoning back-ends. This is a vast area beyond the scope of this paper but an excellent reference on automated reasoning is provided by Harrison [90].

2.4.2.1 Static Analyzers. Static analysis can range from something as simple as a "linter" (a program that enforces coding standards and identifies potentially incorrect patterns) to more fully-featured analyzers. Corporations such as Google [170] and Facebook develop their own static analyzers. FBInfer is an open-source static analyzer for C, C++, Objective-C, and Java

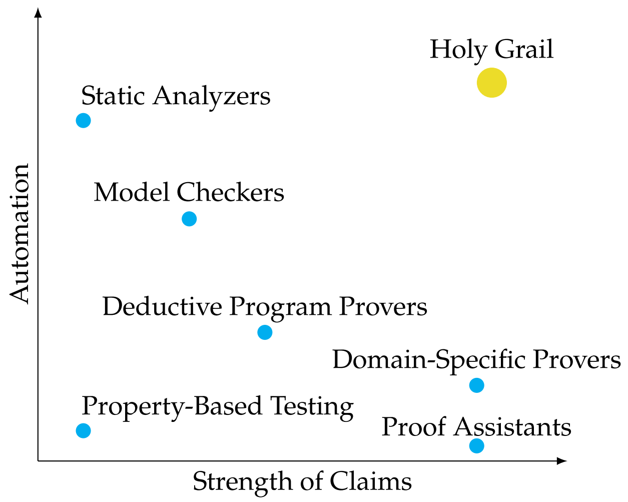


Figure 2. The landscape of formal verification, inspired by Leroy [126]. Up and to the right is good, but the holy grail is unattainable.

Table 2. Categories of formal methods.

Method	Example Input	Example Output	Notable Example(s)
Static Analyzer	source code	dereferencing null pointers	Astreé [51], Cppcheck [131]
SMT Solvers	propositional or 1st-order logic	satisfiable or unsatisfiable	Z3 [62]
Model Checkers	petri nets, büchi automata	liveness or safety property	SPIN [99], Helena [70]
Deductive Program Provers	annotated source code	partial correctness certificate	Why3 [72], Frama-C [52]
Proof Assistants	formal proof	proof certificate	Coq, HOL [83], PVS [124]

supported by Facebook [71]. Google started using FindBugs [8] to analyze Java programs then developed their own static analyzer. France's most successful static analyzer is Astreé [25]. One concern of static analyzers such as Clang or FBInfer is they can produce false positives (valid code is labeled as invalid) or false negatives (invalid code is labeled as valid). Tools which faithfully implement abstract interpretation such as Astreé are sound—no false negatives. Of course, we must temper our expectations because static analysis cannot detect all possible errors. However, a large class of errors such as division by zero, arithmetic overflow, array out of bounds, and many kinds of data races can be checked using static analyzers. Additionally, the ROSE project from Lawrence Livermore National Labs [172] is a source-to-source compiler which facilitates static analysis of source code and binaries.

Predicate abstraction is a form of abstract interpretation wherein the abstract domain is constructed by a user-provided collection of predicates regarding program variables. This is a more robust method of abstraction than loop invariants and these abstractions can generate loop invariants. However, deductive program provers can ultimately prove stronger properties.

The original paper on predicate abstraction uses the PVS theorem prover [86]. However, more modern approaches apply predicate abstraction to analyze languages like Java [75]. The Berkeley Lazy Abstraction Software Verification Tool (BLAST) [92] and Microsoft's SLAM project used to verify code used in device drivers [12] are more examples of predicate abstraction which both take as input C programs.

One prominent static analyzer is Frama-C [52]. This suite of program analyzers (Frama-C calls them plugins) takes as input annotated C code. These annotations are written in a language called ANSI/ISO C Specification Language (ACSL). The program is then run through Frama-C which uses various methods including abstract interpretation, Hoare Logic, as well as deductive verification to analyze various properties of the program.

2.4.2.2 Model Checkers. Model checking is typically separate from the actual program. One common use case of model checking is a machine-readable specification that supplements both a written description and source code implementation. Then, the model checker verifies the behavior. In alignment with the duality of Definitions 2.1 and 2.2, model checkers often consist of a specification language and checking engine. The following examples are for modeling and verifying concurrent and distributed software:

– The modeling language PROMELA and the SPIN model checker [99].

– The modeling language TLA+ and its checker TLC [119].

– Petri Nets [150] and the Helena [70] model checker.

– NuSMV's [41] specification language which is used to describe finite state machines.

2.4.2.3 Deductive Program Provers. Deductive program provers seek to combine static analysis, SMT solvers, and model checkers to prove strong properties of a program without requiring full proof assistants. These tools work by annotating a program with a more complete specification. These annotations are typically embedded in comments in the code. The program with annotations is used to generate *verification conditions* which then are dispatched to various back-ends such as SMT solvers.

An example of this approach is Why3 [27] from INRIA. With Why3, specifications are written in WhyML, an ML-style language extending first-order logic with some common programming bells and whistles such as algebraic data types, polymorphism, and pattern matching. Why3 aims to be a front-end for theorem provers and can also dispatch verification conditions to Coq.

Additionally, Frama-C is not just a static analyzer because of its collection of plugins. It fits the description of deductive program prover because of its reliance on annotation followed by proof dispatch. Another static analyzer that goes beyond just static analysis is SPARK [13]. SPARK is a formally defined subset of the Ada programming language which includes code annotation and can dispatch verification conditions to the SPADE proof checker, whether these are proven automatically or manually. Yet another project from Microsoft Research is Dafny [123] which dispatches proof obligations to other Microsoft projects like Boogie [14] and Z3 [62].

All these layers of specification languages, intermediate representations, and proof checkers can get confusing, even for professionals in the field. To this end, workflows have been developed in order to manage these levels of abstraction and decidability [46].

2.4.3 Human and Computer Time.

Randomized property-based testing such as QuickCheck [42] requires relatively little human time but randomized inputs allow an arbitrary amount of computer time to be spent. Conversely, thinking of specific, known edge cases will take more human time but less computer time. This may be useful for Continuous Integration (CI) scenarios where long-running unit tests can disrupt the workflow of an active project.

The strongest guarantees such as termination typically require proof assistants. On the other hand, unambiguous specifications can also be time-consuming to define. We also note proof assistants are not necessarily fast; a user may have written tactics which take an unreasonable amount of time and thus will require more human guidance to have the proof terminate in a reasonable amount of time. The simplest example of this is writing a tactic that exhaustively checks all proofs consisting of n terms; this is akin to attempting to prove a theorem by picking words out of a hat.

Further automation can be achieved via domain-specific heuristics. For example, Gappa [59] can search through a large number of lemmas in floating-point arithmetic such as $x + 0 = x$ to guide proofs. At the same time, Gappa is completely useless when reasoning about heap memory for example, further demonstrating our quest toward the Holy Grail in Figure 2 is limited.

We outlined three questions a researcher must answer when verifying a program. The first is: what is the strength of the property we wish to verify? We used the word strength here to mean to what degree can a program misbehave while still maintaining our verified property. For example, a nonterminating program is still partially correct but we would probably not consider it correct. In general, the stronger the claim, the more human effort is required to prove this claim. We then listed and briefly described popular tools for verification. Lastly, we emphasize this tradeoff between human time and computer time. This section is in some sense just an explanation of Figure 2.

While the techniques just described all are important methods to verify software, this dissertation does not use deductive program verification or proof assistants. Current research tools have difficulty scaling more automated tools up to large problem sizes; the problem remains unsolved for more formal, less automated tools like proof assistants. Next, we describe some applications of these verification techniques. We focus on intermediate representations (Section 2.5) because of its applicability to binary verification, which we investigate in Chapter III.

2.5 Intermediate Representations

Before discussing verification techniques for intermediate representations/intermediate languages (IRs/ILs) we answer the question: why would we not just reason about high-level languages or binaries instead? For one, IRs may have nicer properties than assembly languages; for example, LLVM's IR is typed and has a formal semantics. The translation from IRs to binary is straightforward so the cost of proving correct translation is not prohibitively difficult. Despite being straightforward, this difficulty is non-negligible. However, one IR can have several different front-ends (high-level languages) and back ends (ISAs). Thus, we get a unified format to reason about a program that generalizes easily to multiple programming languages and architectures, depending on the IR used. Additionally, compiler transformations often operate on source code or an IR itself so we can more easily verify code transformations and compilers by verifying the IR.

2.5.1 Reasoning about Assembly Language. One important contribution to the design of assembly languages which allows better reasoning about programs is *static single assignment* form (SSA), first proposed by Rosen et al. [161]. SSA is widely used in intermediate representations (IR) and most of the IRs we describe throughout this report are in SSA, notably in GCC and LLVM [121]. The defining characteristic of SSA is each variable is only written to once. SSA facilitates many optimizations such as common subexpression elimination. This is accomplished by introducing ϕ-nodes in the control flow graph (CFG). Wherever two or more vertices in the CFG join, we insert a ϕ node by replacing the assignment of a variable with ϕ to indicate the assigned variable can take one of several values. For example, a join point can be at the end of an if-then-else statement shown in Figure 3.

```
// initialization
if $x < 5$ then
    $x = y + 1$
else
    $x = z + 2$
fi
$w = x$
```

```
// initialization
if $x_1 < 5$ then
    $x_1 = y_1 + 1$
else
    $x_2 = z_1 + 2$
fi
$w_1 = \phi(x_1, x_2)$
```

Figure 3. An example of join points in static single assignment form.

Mechanically-verified assembly languages have been researched since 1989 with the Piton assembly language [139]. However, the most popular IR is the language used in LLVM [121], which surprisingly, doesn't have a name other than "LLVM IR." After the success of LLVM, Zhao et al. formalized LLVM's IR with Vellvm in Coq (Verified LLVM) [205]. Key contributions of Vellvm were formalizing the nondeterminism which arises from the fact that LLVM can be underspecified (using the **undef** keyword) as well as allowing for arbitrary memory models.

2.5.2 Compilers. As early as 1967, compilation algorithms have been verified with respect to the source semantics are translated into a target semantics [110]. However, there is still the concern of whether a programmer's implementation matches the algorithm written on paper. In 2000, verified transformations were applied to GCC [144]. However, it wasn't until several years later that a more complete compiler verification story was undertaken by Xavier Leroy.

The most prominent project striving for complete formal verification of a compiler is CompCert, written in Coq [125]. In addition, there is an ambitious project led by Andrew Appel called the Verified Software Toolchain [4] (VST), of which CompCert is one step. In the VST, the goal is the formal verification of the input code (viz. static analysis), the compilation (viz. CompCert), and a runtime system (viz. operating system) to verify a program's behavior. The VST ultimately also aims to support concurrent behavior via shared-memory semaphores.

CompCert and the VST assume the same exact compiler is used on all parts of the program. However, modern software simply is not developed this way because of the heavy reliance on standard libraries coded in multiple languages. Amal Ahmed's research group seeks to address these issues [149]. Other verification efforts for compiler optimizations include the Alive project from Microsoft Research [129], more optimization passes in CompCert [140], and local optimizations using SMT solvers [35].

2.5.3 IRs in Practice. Microsoft Research focuses heavily on verification through domain-specific languages and IRs. For example, CIVL is a concurrent intermediate verification language which extends the Boogie intermediate language for concurrent programs [14]. CIVL looks similar to a markup language for assembly IRs [91].

A project from CMU called the Binary Analysis Platform (BAP) [33] consists of an intermediate language which makes all side effects of a given assembly instruction (such as setting condition codes) explicit. BAP supports subsets of Intel x86-64 and ARM ISAs and can generate verification conditions in a weakest-precondition style specification. BAP can also be used to mine general information about a binary such as instruction mixes. Once the desired postcondition is described, BAP dispatches the verification of the weakest precondition to an SMT solver. One drawback to BAP is its limited support of ISAs. The complete x86-64 instruction set, including with vectorized instructions, consists of thousands of instructions so it is difficult for projects to keep up.

A feature of many binary analyzers is their use of intermediate representations or intermediate languages (IRs/ILs). For example, BAP has its own IL, Valgrind has VEX. These ILs/IRs are often formally specified (as with BAP and Vellvm) to remove any ambiguities. Another benefit of ILs/IRs are the increased portability; support for a new ISA requires only adding a new parser.

A common theme throughout verification tools is that the workflow consists of three steps:

1. Annotation of a program, either hand-written or automatically generated,
2. Transformation of this program into an IR via some semantics such as symbolic execution-style predicate transformers,
3. Transform this IR into a common format such as SMTLib then dispatching to an SMT solver.

In between these steps is often optimization to mitigate the state space explosion problem.

Other work includes verification of an extension of MIPS assembly language [1]. Recently, WebAssembly has been formalized and its type system proved sound in Isabelle [199]. WebAssembly is meant to be a target for high-level languages which can run on web clients and servers while also using the local machine's hardware directly to improve performance. Hardware corporations tend to adopt formal methods much better than software corporations because hardware cannot be modified once it is shipped. For example, ARM is in the process of converting its semantics to a machine-checkable representation [155].

Microsoft Research has a recent project to combine its specification language F* and a subset of the Intel x86-64 assembly language to verify hand-optimized code [77]. Last but not least, we give special mention to Valgrind [145], a tool that consistently surprises people with its features. Valgrind also has static and dynamic analysis, concurrency error detection, memory leak checking, and profiling.

2.6 Floating-Point Arithmetic

2.6.1 Floats, Bits, and the Real Line. Floating-point (FP) arithmetic is the workhorse of scientific computation yet is fundamentally an approximation since it represents an uncountably infinite set, the real numbers (denoted $\mathbb{R}$), in a fixed number of bits. While this approximation is usually good enough, we wish to describe when and how the approximation fails.

The most common encoding of $\mathbb{R}$ is the IEEE 754 standard. We refer to its most recent specification, the 2019 revision [204]. The IEEE 754 standard for floating-point arithmetic is the most ubiquitous standard by far, but we wish to understand other formats in a unified way. IEEE 754 was first standardized in 1985, so before then FP arithmetic varied across hardware. We describe these in Section 2.6.4.

While popular, IEEE floats certainly have their quirks. For example, having the same bit pattern is neither necessary nor sufficient for two IEEE floats to be considered equal. For example, the bit pattern $0 \ldots 0$ and $10 \ldots 0$ represent -0 and $+0$, respectively, and are equal but whose bit patterns are not the same. Conversely, Not-a-Numbers (NaNs) may have the same bit pattern but are defined to be not equal to every other float, including the same NaN bit pattern.

A good introduction to the concerns when representing FP numbers is described in 1991 by Goldberg [80]. However, there are many important concepts from numerical analysis which are useful when describing the intricacies of FP arithmetic. A short introduction to these concepts is described in Section 2.8.

When reasoning about floats, naïvely treating FP properties as instances of a Satisfiability Modulo Theories (SMT) can lead to pitfalls [47]. For example, (2.4) gives an unsatisfiable proposition; proving this proposition unsatisfiable equates to proving if $x \leqslant y$ then it is impossible for $y + z < x + z$ for some small z:

$$\begin{aligned} &(-2 \leqslant x \leqslant 2) \\ \wedge &(-2 \leqslant y \leqslant 2) \\ \wedge &(-1 \leqslant z \leqslant 1) \\ \wedge &(x \leqslant y) \\ \wedge &(y + z < x + z). \end{aligned} \tag{2.4}$$

Treating these variables as vectors of bits ("bit blasting") instead of using their real number properties takes an intractable amount of time with even sophisticated SMT solvers. This example will be discussed further in Section 2.6.5. Broadly speaking, formalizations of FP arithmetic either treat FP numbers as subsets of real numbers or as a collection of bits. While not strictly true (we shall give special attention to a Coq library called Flocq [28]), this provides a good characterization.

Using the "floats as bits" interpretation, bit pattern quirks of IEEE floats can be easily handled but properties of real numbers can be intractable to specify. Using the "floats as reals" interpretation, one can express any theorem of real analysis and determine how FP operations map to these theorems. Libraries instead focus on operations on $\mathbb{R}$ and how well floats approximate these operations via the semantics of rounding. Conversely, real analysis is not understood by the humble transistor and real-world code may do strange things to the bits of a float while still being correct according to real-number semantics.

2.6.2 Floating-Point Representations. A floating-point number is represented as

$$\pm k\beta^{p}$$

where $\beta > 1$ is the radix, p is the exponent, and k is the mantissa. To make notation a bit[2] clearer we use $\beta = 2$, though $\beta = 10$ is also widely used.

While we could pick any range for k, in reality either $0 \leqslant k < 1$ or $1 \leqslant k < 2$. This means we have $k = (1.k_1k_2k_3 \cdots k_n)_b$ or $k = (1.k_1k_2k_3 \cdots k_n)_b$. Here the subscript b indicates the number is expressed using a binary radix. Since we know the first bit is either 0 or 1 these are omitted with a binary representation and we say a number is *normalized* with an implied 1 and *subnormal* with an implied 0.

2.6.3 IEEE 754. By far the most common floating-point standard is the IEEE 754-2019 standard, which is implemented on nearly every modern processor. With IEEE floats, the exponent is an e-bit number. Values of e are shown in Table 3. Given some bit pattern representing an unsigned integer p, we compute the actual exponent as $p - \mathit{bias}$, where $\mathit{bias} = 2^{e} - 1$. One might think this means we have representable exponents in the range $\{-2^{e-1}-1, \ldots, 2^{e-1}\}$ (for 32 bits this is $\{-127, \ldots, 128\}$) but in reality, this range is 2 smaller because an exponent pattern of all 1's signals Infinity (Inf) or Not a Number (NaN) and an exponent

[2]pun intended

Table 3. Some important values for floating-point representations.

Name	Bits	Radix	e	m	$e_{\min}$	$e_{\max}$
IEEE	n	2	e	m	$-2^{e-1}+2$	$2^{e-1}-1$
half	16	2	5	10	−14	15
single	32	2	8	23	−126	127
double	64	2	11	52	−1022	1023
quad	128	2	15	113	-16,382	16,383
posit	n	2	e	$n-3-e^*$	$-(n-2)\times 2^e$	$(n-2)\times 2^e$
posit	32	2	3	26^*	−240	240
posit	64	2	4	57^*	−992	992
1750A	32	2	8	16	−128	127
1750A	48	2	8	32	−128	127

* Tapered precision; maximum is shown here

pattern of all 0's signals a subnormal number. That is, the range of exponents for exponent size e is

$$\{-2^{e-1}-2,\ldots,2^{e-1}-1\}.$$

For example, with IEEE 754 half-precision, the following indicates the smallest normalized float and the largest subnormal float.

$$0\ 00001\ 0000000000 \mapsto 2^{-14}\times 1.0_b$$

$$0\ 00000\ 1111111111 \mapsto 2^{-14}\times 0.1111111111_b.$$

IEEE 754 is implemented such that the result of an FP operation should be the result of the operation applied to real numbers (i.e., with infinite precision), then rounded to fit in the correct number of bits. Practically, the rounding aspect is the most difficult to reason about and is the cause of the most insidious FP bugs. IEEE 754 allows both normalized and subnormal values, though subnormal numbers have what is called *gradual underflow* by sacrificing lower precision to represent numbers closer to 0.

The IEEE standard defines five exceptions:

1. Invalid Operation, e.g., $0.0\times\infty$
2. Division by Zero
3. Overflow, i.e., to either $+\infty$ or $-\infty$
4. Underflow, i.e., to $+0$ or -0

5. Inexact, e.g., $\sqrt{2}$

The GNU C library manual describes how they handle this [188]. Typically if one of the five exceptions occur, a status word is set and the program continues as normal. However, one can override this to throw a `SIGFPE` exception which can be trapped.

Another subtle feature of IEEE 754 is the difference between *quiet* and *signaling* NaNs, (qNaN and sNaN). Distinguishing between these types of NaNs is hardware-dependent but are typically differentiated by the most significant bit of the mantissa. For example, the canonical sNaN and qNaN for RISC-V are:

```
#define qNaNf 0x7fc00000
#define sNaNf 0x7f800001
```

Thus, there is a wide range of valid signaling and quiet NaNs ($2^{52} - 1$ different bit patterns are valid NaNs for double-precision floats).

While this is an IEEE specification, not all implementations (for example, Java) make this distinction between quiet and signaling because it requires hardware support and the five exceptions paired with traps sort of already handle this. Additionally, the behavior of sNaNs is quite messy because the compiler may optimize away instructions which would generate these sNaNs, thus not throwing an exception.

Because of this, one might ask the question "Why would I even bother with signaling NaNs?" One reason would be to catch FP errors earlier on. For example, one could set an uninitialized FP value to a sNaN to raise an exception in all cases rather than potentially propagating a garbage value. Signaling NaNs allows catching of some operations in which qNaNs do not propagate; for example

$$\min(x, \text{NaN}) = \min(\text{NaN}, x) = x$$

for all x which aren't NaNs (including ∞!).

The last and most important concept with IEEE floats is the rounding mode. IEEE 754 specifies five rounding modes:

1. Round to nearest, ties break to even

2. Round to nearest, ties away from zero

3. Round toward zero
4. Round toward positive infinity
5. Round toward negative infinity

The default behavior is almost always round to nearest, ties break to even, which we write as round-to-nearest.

2.6.4 Other Floating-Point Formats. We briefly describe other FP formats which seek to either address issues with IEEE 754 floats, simplify their implementation for performance, or were invented before IEEE 754's standardization in 1985.

Posits are a new FP format invented by John Gustafson with strong claims for better performance and accuracy without actually being implemented in hardware nor much empirical evidence [87]. However, it is an interesting format which if the claims are correct could usurp IEEE 754 as the dominant FP format.

MIL-STD-1750A is an open instruction set architecture describing an instruction set architecture including two FP formats [194]. One interesting feature of the floats of MIL-STD-1750A is since all FP values are subnormal, to prevent multiple representations of the same number all mantissa must be in the range $[-0.5, 0.5]$. This means a large number of bit patterns are invalid floats.

Hardware manufacturers in the interest of performance may simplify aspects of IEEE floats. For example, NVIDIA allows flags which do not give full accuracy for division and square root as well as flushing subnormal numbers to zero [200]. Additionally, on a GPU traps for FP exceptions are typically not supported.

Machine learning hardware may implement nonstandard FP formats. For example, one 16-bit format is called a *bfloat* (brain float) which has 8 bits for the exponent and 7 bits for the mantissa (contrasted with 5 and 10 for IEEE half-precision) [184]. Microsoft has developed its own floating-point format too [163].

2.6.5 Formalizations of Floating-Point Arithmetic. One early formal description of FP arithmetic was introduced in 1989 [18] defined using the specification language Z [180] but this formalization is not maintained.

Another formalization of IEEE 754-2008 floats is an extension to the SMT-Lib Standard done by Brain, Rümmer and Wahl [165, 32]. This extension supports exceptions, the usual mathematical operations, and rounding modes. However, SMT representations are limited because they essentially only think of floats as their bit-level representations and not how they relate to $\mathbb{R}$. This means an SMT solver when determining bounds of some operation has no good way to estimate them and in many cases must exhaustively check all FP values. For example, it is difficult to represent a property like $x + y = y + x$ for FP arithmetic. Another example that SMT solvers struggle with was previously shown in Equation (2.4).

To address some of these limitations, Brain et al. developed an Abstract Conflict-Driven Clause Learning (ACDCL) [31]. This method is based on choosing an abstract domain over which to apply the CDCL algorithm [132]. ACDCL is one part of an effort to unify abstract interpretation with SMT techniques. As it applies to FP arithmetic, ACDCL creates abstract domains for interval arithmetic. So for example, the formula $x = y + z$ may restrict $y, z \in [0, 10]$. Thus, interval arithmetic states $x \in [0, 20]$. This also cannot solve even a simple problem such as (2.4) easily because the abstract domain is not precise enough. Thus, the community requires more expressive theories about FP operations.

Formal verification of FP arithmetic and numerical algorithms primarily comes from the Toccata research group [100]. One example is creating a specification of FP numbers in Coq, called Flocq [28]. Boldo & Melquiond also wrote a book describing verification of some numerical algorithms using Flocq [29]. Flocq allows reasoning about arbitrary rounding modes and is generic enough to define other FP formats. However, there is no concept of overflow since exponents can be of arbitrary size in Flocq. The authors acknowledge this and mention it is easier to reason about overflow separately from correct rounding and error analysis. However, this still distinguishes Flocq as a library on the side of "floats as reals."

In addition to Flocq, there are PFF, Gappa, and Coq.Interval which accomplish related goals. PFF focuses on arithmetic without rounding errors and has been subsumed by Flocq [57]. Gappa more closely resembles numerical code written in a C-like language and is designed to be easily machine-checkable [59, 60]. Coq.Interval is a library to formalize interval arithmetic and is also mostly subsumed by Flocq.

The limitations of SMT solvers when reasoning about real numbers caused Conchon et al. [48] to develop a method which uses both Gappa and the Alt-Ergo SMT solver [115].

However, this efficiency comes at the cost of automation; while Gappa can reason about high-level properties of FP arithmetic (e.g., $x + y = y + x$), Gappa typically requires source code annotation.

While the most mature package for reasoning about floats is Flocq, hardware manufacturers have been early adopters of formal methods and FP programs are no exception. Work at AMD [169] using the ACL2 prover focuses on register transfer logic. Additionally at Intel, Harrison worked on FP verification using HOL Light [89]. However, these works are not publicly available so we do not know if their projects are still active.

2.6.6 Notation. It may also be useful to distinguish between interpretations of FP values and $\mathbb{R}$. To the best of our knowledge, the following set describes all possible FP values. We use some notation from Flocq [28] such as f2r but most of this formalization is novel in an attempt to unify "floats as bits" and "floats as reals:"

$$\mathbb{F} = \{\text{NaN}, 0, -0, -\infty, \infty, \pm\infty\} \cup \{\beta^e \times k_0.k_1k_2 \cdots k_n \mid n, e, \beta \in \mathbb{Z}, k_i \in \mathbb{Z}_\beta, \beta > 1\}. \tag{2.5}$$

Note that both mathematically and practically speaking $\mathbb{F} \not\subseteq \mathbb{R}$. For any FP format f which is represented using n bits, we write an FP number x as a string of n bits and so $0 \leqslant x < 2^n$. Henceforth we write $x \in \mathbb{Z}_{2^n}$.

The second portion of (2.5) is a subset of the rational numbers. This precludes the possibility of irrational numbers like $\sqrt{2}$, or π from being represented exactly. This is an issue, because one would wish FP arithmetic in some sense remember its rationality, such as $\sqrt{x} \times \sqrt{x} = x$. Alas, this does not in general hold. As we have seen with the division example in Table 1 we cannot even guarantee $\frac{1.0}{x} \times x = 1.0$ for all finite nonzero $x \in \mathbb{F}$. However, approximations must suffice both for efficiency and decidability. That not all real numbers are not computable has been proven more generally by a relatively distinguished computer scientist [193].

We now introduce some mathematical formalizations similar to Flocq. We begin with some floating-point representation f. While Flocq does not concern itself with bitwise representation, we roll this into f. One motivation for this is hardware; specifications such as MIL-STD-1750A specify FP operations with respect to masking, shifting, and adding bits. Similarly, built-in hardware operations such as tests for equality, absolute value, or negation could be used on FP values as potential optimizations. We present a list of the FP representations we consider (the set of f's) in Table 4.

We also need three more pieces of notation to understand f. Notice in (2.5) we do not specify an n. We do this to ensure a float *can* be represented by some finite number of bits. This is to allow for the development of rounding schemes which may need extra bits to compute the correct rounding but also for generality.

If we must specify exactly the set of representable floats for a given f we write $\mathbb{F}_f$. For example, with 32-bit posits

$$\mathbb{F}_{\text{posit32}} = \{\pm\infty, 0\} \cup \{2^e \times 1.k_1k_2 \cdots k_{26} \mid k_i \in \mathbb{Z}_2, e \in \{-240, \ldots, 240\}\}$$

because there is no NaN, only a single value for infinity, and mantissas have at most 26 bits. In reality, posits have *tapered precision* which means for larger e the mantissa is shorter (thus requiring the a portion of the least significant k_i to be 0) but this is a sufficient description for illustrative purposes.

Next, we need some function interpreting binary as a float:

$$\text{b2f}_f : \mathbb{Z}_{2^n} \to \mathbb{F}$$

and a function interpreting a float as a real

$$\text{f2r}_f : \mathbb{F} \to \mathbb{R}.$$

These may be quite complicated and messy. For example, b2f_f may be neither total nor injective. In IEEE 754, many bit strings map to NaN and in MIL-STD-1750A there are many bit patterns which are not valid floats. On the other hand, f2r_f is typically neither total (NaN does not map to a real) nor injective ($+0$ and -0 map to $0 \in \mathbb{R}$).

Along with f we have operators on floats denoted with a subscript: $<_f, =_f, \times_f$, etc. We expect these to behave similarly to their corresponding operators on real numbers but there are important differences.

Lastly, one may wish to have inverses of b2f and f2r. This is in general impossible because they are usually neither injective nor surjective. However, analysis of FP arithmetic correctness requires us to think of how $\mathbb{R}$ embed into floats. This is done through the various *rounding modes*. For some rounding mode r associated with an FP format we define

$$\text{round}_r : \mathbb{R} \to \mathbb{F}.$$

Table 4. A characterization of three floating-point representations.

Standard	NaN	Infinite	Subnormal	Zero(s)
IEEE	Yes	$+\infty, -\infty$	Yes	$+0, -0$
Posit	No	$\pm\infty$	No	0
1750A	No	No	Always	0

In contrast with f2r, all rounding functions are total. For practical reasons, round_r has a canonical binary representation for each float to which it rounds. To keep notation cleaner, we overload round_r to map both into $\mathbb{F}$ and $\mathbb{Z}_{2^n}$; the former codomain is preferred when reasoning about floats more theoretically and the latter when you actually need to push some bits around.

2.6.7 Properties of Floating-Point Representations. We present some examples of properties that help with reasoning about FP numbers. A small piece of notation is 2C means "two's complement." Additionally, while we typically write + these properties are also desired for multiplication.

1. Monotonicity: if $x, y \in \mathbb{Z}_{2^n}$,

$$x <_{2C} y \implies \text{f2r}(\text{b2f}(x)) <_{\mathbb{R}} \text{f2r}(\text{b2f}(y)).$$

   Intuitively, we wish typical two's complement integer comparison operators to work with FP values.

2. Correct Rounding: this a constraint on the results of any FP operation that the last digit is rounded so that it is closest to the true answer. For some inputs, this requires extra bits of precision. In general, this is a challenging undertaking, but most FP hardware operations and some libraries are correctly rounded for all inputs [58].

3. Sterbenz Lemma: informally, if two numbers are sufficiently close then we can exactly represent their difference (without rounding error).

   Formally, Sterbenz Lemma holds for a given FP scheme f if

$$\forall x, y \in \mathbb{R},\ \frac{y}{2} \leqslant x \leqslant 2y \implies$$
$$\text{round}(x - y) = \text{round}(x) -_f \text{round}(y).$$

4. Associativity: if $x, y, z \in \mathbb{F}_f$,

$$(x + y) + z = x + (y + z)$$

   This does not hold in general, but we might want to know what further conditions must hold on x, y, z to ensure associativity, either with addition or multiplication.

In summary, reasoning about FP arithmetic beyond just the well-behaved cases is difficult. There exists no complete unification of the dual interpretation of "floats as bits" and "floats as reals." However, packages such as Flocq are a step in the right direction and we expanded on some formalizations given in Flocq as a starting point to future work.

Next, we mention a concept which goes hand-in-hand with scientific computing applications which consist primarily of floating-point arithmetic: parallelism.

2.7 Parallelism and Floating Point

We describe efforts toward verification of parallel floating-point programs. The first approach to describing parallelism formally was by Robert Keller in 1976 [111]. He described a program as a set of states Q and a transition relation $\rightarrow$, a binary relation on Q. Importantly, for some $q \in Q$, there may be many q' such that $q \rightarrow q'$; these represent the interleavings of execution traces. The groundwork laid by Robert Keller has influenced verification techniques such as TLA+ [119].

Existing symbolic execution tools such as KLEE have been used to verify the correctness of vectorized code using a technique called symbolic crosschecking [45]. With this approach, the vectorized code is expressed as symbolic operations, then checked against a serial equivalent. This approach is conservative because it ensures the exact same operations are performed with SIMD code and symbolic code. Interestingly, Collingbourne et al. claim there exists no floating-point constraint solvers available. However, in 2001 Michel et al. presented a floating-point constraint solver [137]. Collingbourne may not have noticed or considered the work by Michel because of its lack of mechanization and focus on test pattern generation. Another related project is called FloPSy, a floating-point constraint solver for symbolic execution, the underlying strategy of KLEE [118]. Regardless, there is not a large amount of work being done in this area.

Instead, researchers interested in performance focus more on higher-level error analysis rather than efficient translation of SIMD operations. For examples of such efforts, we direct the reader to the Correctness Workshop at Supercomputing [116] and the US Department of

Energy's summit on Correctness for HPC [82]. A common strategy for SIMD parallelization is through compiler directives such as OpenMP or Intel Thread Building Blocks (TBB). Blom, Darabi, and Huisman create a separation logic-based approach to prove independence between loop iterations [26].

Intermediate representations (IRs) are at a surface level attractive for handling verification of parallel programs: heterogeneity of hardware and programming models should hopefully not require different verification techniques when the underlying concepts (happens-before relations, synchronization, atomic operations) are the same. In practice, IRs are often less mature than source programs. Additionally, the implementer must know yet another programming model. One attempt at using IRs is a symbolic execution-based model checker called CIVL [206]. This compiles various parallel programming paradigms into an intermediate representation, CIVL-C, as long as the input program is written in a C-like language such as CUDA-C, MPI, OpenMP, and can handle hybrid parallelism. This should not be confused with the aforementioned CIVL project from Microsoft Research used to reason about the correctness of concurrent programs [91].

2.8 Floating-Point Error Analysis

Floating-point error analysis has its roots in numerical analysis and so predates even the first mechanical computer, the Z1, built in 1938 [160]. Numerical analysis existed before FP error analysis, and even if FP error were to completely disappear, numerical analysis would still be a vibrant field. We define a few key terms necessary to understand numerical analysis; for a more complete exposition there are several good books on the topic [171, 153].

Even so, FP error analysis has become a field of study in its own right. Numerical analysis is the study of mathematical processes for numerical computation (as opposed to symbolic computation) of analytical objects. For example, Newton's method iteratively builds an approximation of a solution to $f(x) = 0$ for some function f. A result of numerical analysis is the rate at which Newton's method converges. Such approximation errors are inherent in numerical algorithms and allow us to build approximate solutions to the many mathematical problems which have no closed-form solution. As an aside, numerical linear algebra, such as matrix factorization, is considered part of numerical analysis despite there oftentimes having closed-form solutions. This is because the large number of operations required may either call for approximate methods or issues such as numerical stability arise.

FP error analysis instead focuses on the implications of limited-precision representations of real numbers. For a simple distinction, 1/10 is a quantity which can be considered as part of a numerical algorithm but is not exactly representable in binary FP representations. That said, the ubiquity of numerical analysis done on digital computers means these two are closely related. With the following research, we will see the actual behavior of FP arithmetic is quite subtle; error from FP arithmetic can range from exactly none to effectively infinite. Determination of *where* in this space we operate for a given numerical program is challenging and the focus of this section.

In the last decade, there has been a resurgence in floating-point (FP) error analysis tools. The basic idea is one of abstract interpretation: researchers are modeling the complex behavior of floating-point error analysis with increasingly sophisticated error analysis techniques.

One concept we wish an error analysis tool to have is *soundness*. This is required for the theoretical properties of abstract interpretation, stating that the actual (concrete) behavior must be a *refinement* of the abstract model. Put a bit more formally, given a program P with some floating-point return value $\hat{x}$ corresponding to a real number x, then an abstraction of the behavior of P is an interval I such that $x \in I$. For example, we may soundly approximate $\sin(x)$ by $I = [-1, 1]$ for any real input x.

2.8.1 Types of Error. Before we describe how to do error analysis, we must first describe what error means. Error may be defined many different ways but we present three.

Definition 2.3. *Let $\hat{x}$ be an approximation of x. Then the* absolute error *is* $|\hat{x} - x|$.

Definition 2.4. *Let $\hat{x}$ be an approximation of x. Then the* relative error *is*

$$\left| \frac{\hat{x} - x}{x} \right|.$$

In general, relative error is more informative since it allows comparison across different magnitudes. The drawback of relative error is it is not defined for $x = 0$. A useful analogy is to think of scientific notation versus financial notation. With scientific notation, the important value is the number of *significant digits*. Relative error is a way to measure this, so the number of significant digits can be tracked across computations of varying magnitudes. For financial calculations, oftentimes the precision is taken to be a mill, or \$0.0001. Here, the absolute error would be measured in mills, regardless of the magnitude of the computation.

From Goldberg [80], we describe the accuracy of an FP operation in terms of *units in the last place* or *ulps*. Units in the last place is a common way to describe error relative to the highest possible precision. If a floating-point approximation is the closest of all FP numbers, then it has at most 0.5 ulps. As an example, consider an imaginary FP representation with four digits of precision. The running example has a true result of 1.100 while the computed result is 1.125. The *relative error* of this computation is $\left|\frac{1.125-1.1}{1.1}\right| \approx 0.023$. A single unit in the last place for 1.100 taken to four digits is 0.001, and so the error is also 25 ulps. The absolute error is $|1.125 - 1.1| = 0.025$.

More formally, we use Goldberg's definition of ulps [141, Def. 2.5].

Definition 2.5. *Let* $f = d_0.d_1d_2d_3 \ldots d_{p-1} \times 2^e$ *be a floating-point number with precision* p *and minimum exponent* $e_{\min}$. *Suppose* f *represents a real number* x. *Then the error from representing* x *by* f *is*

$$\left| \frac{f}{2^e} - \frac{x}{2^{\max(e, e_{\min}) - p + 1}} \right|$$

units in the last place.

This formula for the size of the denominator is somewhat complex, to correctly handle subnormal FP numbers.

2.8.2 Challenges with Floating-Point Error Analysis. A typical model of floating-point provides bounds on the relative error of a given operation. That is, given some real operation $\cdot$ and its corresponding floating-point operation $\odot$,

$$x \odot y = x \cdot y(1 + e), \text{ where } |e| < \varepsilon.$$

This is often known as "machine epsilon," as indicated in Table 3. Additionally, using the *round-to-nearest* mode in IEEE 754, we may instead bound this by $\varepsilon/2$. Because of the ubiquity of round-to-nearest, in some literature ε is defined as our $\varepsilon/2$; the choice is ultimately arbitrary as long as the authors are consistent.

While this property holds for all normal floating-point numbers and some operations, this error is insufficient for many subnormal computations. Chapter V dives deeper into this discrepancy. A complete table of these results is given in Table 5 which can be derived from the constants in Table 3.

To see a small example of the challenges when dealing with FP error analysis, consider the humble multiplication between two single-precision floats. Figure 4 picks 200 small single-precision floats, each spaced precisely 10,000 ulps apart (that is, there are 10,000 floats between each sample). With this, a false trend emerges which looks like the error increases along with x. In reality the picture is more complex. An exhaustive testing for the same range is shown

Table 5. Important floating-point constants as shown by Goualard [84].

Format	$e_{\min}$	ε^*	$\lambda^\dagger$	$\delta^\diamond$
IEEE single	-126	2^{-23}	2^{-126}	2^{-149}
IEEE double	-1022	2^{-52}	2^{-1022}	2^{-1074}

* machine epsilon
† smallest positive normal number
$^\diamond$ smallest positive subnormal number

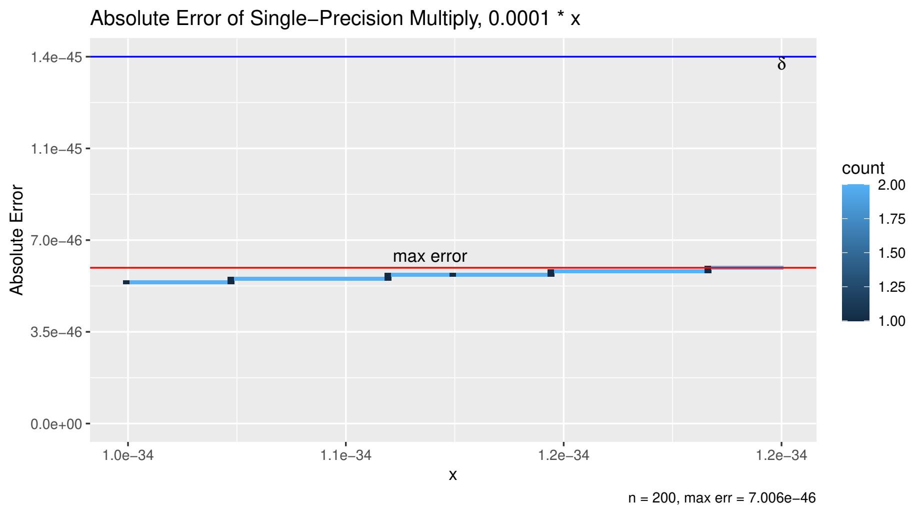


Figure 4. Plotting absolute error for multiplication of 0.0001 $\times$ x for 200 nonrandomly chosen floats each spaced 10,000 ulps apart.

in Figure 5. This plot shows error of multiplication ranging from 0 to $\delta/2$ (not δ, because the rounding mode is set to round-to-nearest). That is, for the given range of x, every single float is checked. From this, we observe that depending on precisely which inputs you use, false patterns may emerge. This is the main thesis of critiques of probabilistic floating-point error analysis [109].

Since exhaustive checking is out of the question for any reasonable input size, more creative methods must be used. The next section describes some tools to address these challenges.

2.8.3 Tools for Floating-Point Error Analysis. A research group led by Gopalakrishnan at the University of Utah develops many useful tools for FP error analysis. Beyond just exhaustive checking, work from Chiang and others at Utah [39] provide an intelligence search to find high-error FP inputs.

Following this, one prominent FP error analysis tool is FPTaylor [178], which uses second-order Taylor series approximations of FP error, then runs a global optimization algorithm on the

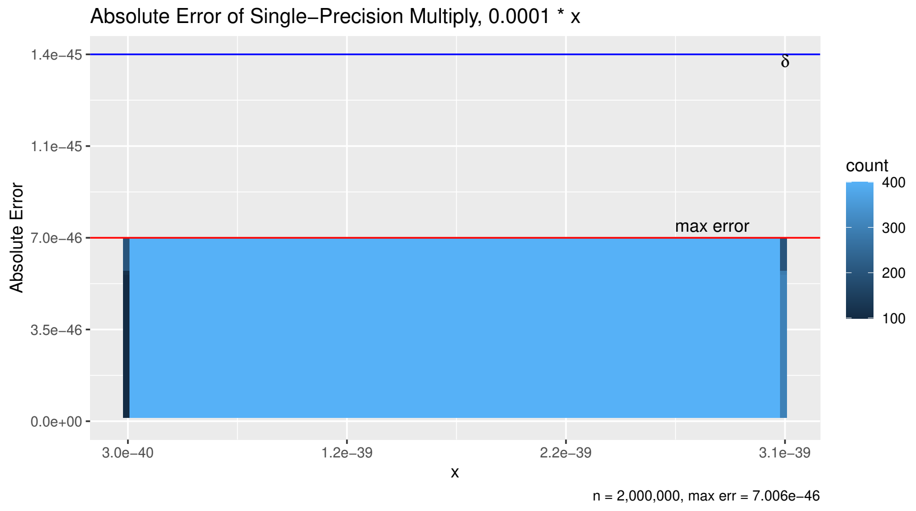


Figure 5. Plotting absolute error for multiplication between 10^{-4} and two million small, consecutive single-precision floating-point numbers.

expressions to find the maximum error. Daisy is another tool which calculates the error of an FP expression but uses SMT solvers and binary search to limit the search space [55]. FloVer [19] accomplishes a similar task, but also provides proofs of correctness in Coq or HOL4. One issue in common with these tools is their lack of scalability. These tools are performing global optimization over large state spaces. For example, brute-force checking of a binary floating-point operation with double precision requires searching a state space of size $(2^{64})^{64}$; the situation only gets worse for more inputs. The fact that they are able to intelligently, exhaustively check error for more than a single operand is impressive; however, these tools do not scale to more than 100 or so floating-point inputs. A more recent tool from 2020 is Satire [56], which is related to FPTaylor but uses first-order Taylor-series approximations and a technique called *path strength reduction* to more aggressively simplify the error terms. This sacrifices some precision in order to scale to larger expressions.

Another related project is Precimonious [164]. This is one of a family of tools which concern themselves with FP representation. Specifically, these tools can be used to discover which encodings of particular FP values should be used. For example, many values may need to be only single-precision while accumulators may be double precision. This is beyond the scope of this work, but we direct the interested reader to the cited materials in the excellent papers from Rubio-Gonzalez, Laguna, and others [117].

2.9 Conclusion

We presented a survey of verification techniques for low-level programs. By low-level, we mean code written in binary, assembly, intermediate representations and programs which interact directly (or nearly directly) with hardware. We began by introducing the logical notation used in formal methods (Section 2.1). In Section 2.4, we listed formal methods tools and how they are used. These provided the background information to explain three applications of formal methods, each to important abstractions in computer science. The first was intermediate representations (Section 2.5), the second was floating-point arithmetic (Section 2.6), and the final was parallelism (Section 2.7). Key abstractions such as high-level programming languages, compilers, and floating-point arithmetic have allowed programmers to manage the incredible complexity of modern hardware and software. In this chapter we provided a more complete picture of how computer scientists reason about the correctness of programmers on programs which do not use these critical abstractions.

2.9.1 Other Formal Methods Surveys. Formal verification encompasses a wide range of disciplines including mathematical logic, type theory, software engineering, and programming languages. A survey of formal methods was published in 2009 [203]. A brief introduction to formal proofs is given by Hales [88]. A previous area exam by Johnson-Freyd focuses on formal methods using temporal logic [104]. Temporal logic refers to logical systems whose propositions are temporally dependent and properties. For example, a temporal invariant for a job scheduler would be, “for all n less than the total nodes of a cluster, a job of size n in the queue will eventually be scheduled.”

With large codebases, it often becomes intractable to formerly verify behavior. In these cases, hybrid techniques such as automatic test pattern generation can be employed. A survey of these approaches is provided by Bhadra et al [20]. Additionally, Sandia National Laboratories has published a survey of existing verification tools for both hardware and software [6].

2.9.2 Formal Methods in Practice. One complaint about formal methods in general is it is too expensive, either in computer or human time [112]. One factor in this complaint is many approaches to formal methods are either exponential, such as checking satisfiability, or undecidable, such as proving total correctness. While not considered formal methods, even the effect of statically versus dynamically-typed languages on productivity and code quality is unclear [154] and has a limited amount of research. A collection of other surveys are described by Luu [130].

Software engineering and formal methods are ultimately human endeavors but to the best of our knowledge, there is no academic study relating code quality, human effort, and the degree of formal methods used. Evidence of its efficacy can be found in the avionics industry which is heavily dependent on formal methods, both in practice and by regulation [126]. That huge companies such as Airbus, AMD, and Intel, as well as the French and United States [2] governments support formal methods hints not only at its profitability but of its value to society.

Slowly but steadily, the percolation of computing into our lives is causing computer scientists and software engineers alike to realize the importance of writing correct software and formal methods' central role in achieving this goal. The following chapters describe our contributions in formal methods. We described many techniques for verifying software, but each of these is advanced areas of study on their own. These concepts are important to keep in mind as different strategies depending on the problem domain. The remainder of this dissertation uses two of the static analysis techniques described in this chapter to verify low-level programs: symbolic execution 2.3.2 and abstract interpretation 2.3.1.

CHAPTER III

QUAMELEON: A LIFTER AND INTERMEDIATE LANGUAGE FOR BINARY ANALYSIS

Nota Bene: This work is based on previously published co-authored work [151], the other co-authors worked at Sandia National Laboratories. The following lists their contributions:

– Samuel D. Pollard programmed the majority of the source code for the version of Quameleon described in the dissertation and the published work. This includes all of the semantics of the M6800 instruction set architecture, most of the testing framework and test cases, and most of the implementation details of QIL. Samuel also was the main author of the paper which included the majority of the writing and designing of all figures and example programs. Samuel also presented the work in person at the SpISA workshop.

– Philip Johnson-Freyd oversaw the overall language design of QIL and Quameleon, contributed Haskell code, described the semantics of the lightweight dependent types for bit vectors in Haskell that were integrated into Quameleon. Dr. Johnson-Freyd provided feedback for the paper and its figures as well as wrote most of the design for the different encodings of QIL (nominal, finally tagless, and de Bruijn).

– Jon Aytac contributed Haskell code and programmed semantics for other ISAs not mentioned in this chapter.

– Tristan Duckworth contributed Haskell source code for the project, generalized the language with which we describe ISAs, and implemented semantics for other ISAs as well as the interface between QIL and angr.

– Michael J. Carson provided software engineering expertise including interfaces between QIL and angr, and continuous integration/continuous deployment.

– Geoffrey C. Hulette provided general design directions and feedback for the project.

– Christopher B. Harrison was the principal investigator and contributed overall guidance on how this research aligned with Sandia's mission goals and led weekly status updates and overall design direction.

3.1 Introduction

Quameleon is a binary analysis framework: its input is a program in binary or assembly language and its output is some high-level analysis. We accomplish this by transforming (or *lifting*) binaries into an intermediate language (IL), with which we can perform various optimizations while also providing a single interface for analysis back-ends. In this chapter, we primarily describe QIL, the Quameleon Intermediate Language (pronounced "quill").

Returning to this dissertation's main question, Quameleon uses high-level programming features of the type system in Haskell, as well as encoding semantic information about each assembly language to give insight into the behavior of binaries. This analysis is done statically, but Quameleon also provides a concrete execution framework.

We write the semantics of a target machine language in an embedded Haskell DSL, then generate analyzable QIL from target programs. QIL features a simple type system designed for assembly languages, with types for sized bit vectors, code pointers, and memory locations; polymorphism and genericity are limited to the meta (Haskell) level. QIL's most significant limitation is it assumes a Harvard architecture; code and data are separate and self-modification is forbidden. We plan to address this limitation in future work.

This work describes the structure of Quameleon. We describe its front-ends, which provide the lifting of binaries into QIL, the design of QIL, and its back-ends.

Current back-ends include a bridge to angr [176] and a concrete executor. Quameleon supports multiple ISAs, but in this chapter we show examples entirely from the beautifully simple Motorola M6800 ISA.

3.2 Disassembly and Lifting

The first part of our binary analysis work is to *lift* a binary into an intermediate representation. Essentially, this means exposing the semantic structure of the binary at a higher level of abstraction.

Our use case is similar to tools like Ghidra [143] and IDA [93]; this means that, from scratch, the process for analyzing the binary consists of three main steps.

1. Creating a machine-readable specification of the ISA. We accomplish this by creating a Haskell data type and associating a semantics with it.

2. Specifying how to disassemble individual instructions in binary format into this data type.

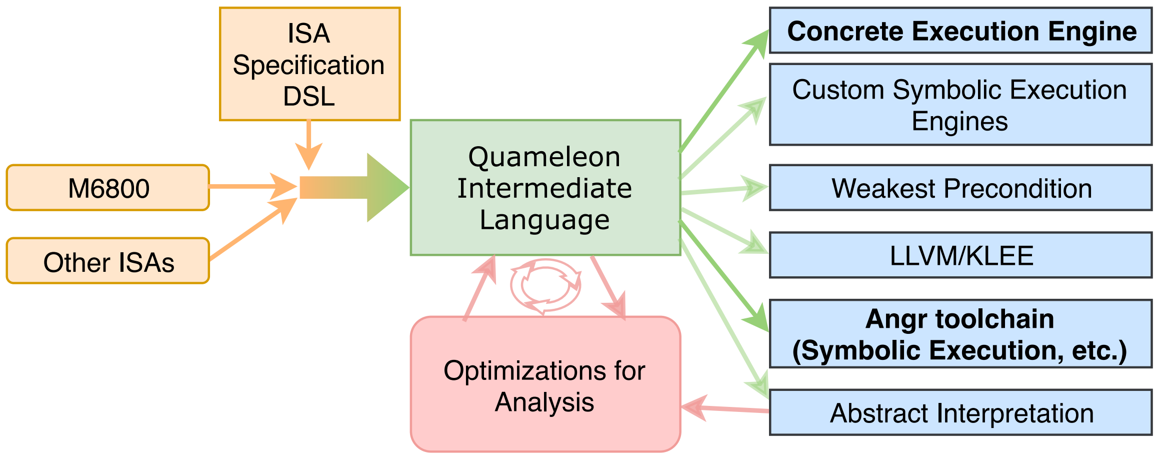

ISA Specification DSL
M6800
Other ISAs
Quameleon Intermediate Language
Optimizations for Analysis
Concrete Execution Engine
Custom Symbolic Execution Engines
Weakest Precondition
LLVM/KLEE
Angr toolchain (Symbolic Execution, etc.)
Abstract Interpretation

Figure 6. Quameleon pipeline. The bolded blue boxes are the currently supported back-ends.

3. Understanding the binary formats used for the ISA and how to separate data from instructions.

One complication is a binary may be self-modifying. This means there is no *a priori* way of disassembling a binary in all cases. We ignore this complication and instead Quameleon greedily disassembles each instruction it knows.

3.2.1 Lifting an Instruction. Lifters to QIL are fairly straightforward thanks to Haskell's type system, the parametricity of QIL, and the simple design of (most) assembly instructions. Usually, specification of the behavior of many instructions can be accomplished in just a few lines of Haskell. As an example, consider the implementation of the logical and operation below.

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

This consists of accessing the register `ra`, a memory location `op`, performing a logical and, writing the result, setting status flags (some were elided for brevity), and finally branching to the next instruction. We note the `andBit` operation is generic in the size of the input; this allows significant code reuse and static checking that operands are well-formed.

3.3 QIL: Quameleon Intermediate Language

QIL is an intermediate language designed to capture the semantics of binary programs for a wide variety of architectures while having an easily formalized semantics.

We designed QIL from scratch so we could provide useful programming language features to ensure strong static type guarantees, provide ease of analysis, and facilitate code transformations. One interesting feature of QIL is its bit vector abstraction which provides statically typed bit vector fields and widths for any bit width. This allows greater code reuse in contrast with other ILs which have separate instructions for different bit widths.

3.3.1 Encoding QIL in Haskell.

We provide three encodings of QIL:

1. a nominal encoding useful for optimization;
2. finally tagless encoding good for code generation; and
3. de Bruijn[1] index encoding to more easily translate representations.

3.3.1.1 Final Encoding.

The QIL encoding primarily used when writing a lifter or back-end will be the "final" (short for "finally tagless") one. Here the idea is that QIL operations are represented as methods of a typeclass, and QIL variables are represented as Haskell values, in a higher-order abstract syntax (HOAS) style, making variable name handling automatic. The final encoding makes QIL feel like Haskell. Conversely, writing a QIL interpreter simply amounts to implementing some typeclasses.

The primary abstractions in QIL are "Locations," "Values," and "Labels". A back-end might implement these in various ways. For instance, our concrete interpreter implements values as bit vectors, but a symbolic executor would probably implement them as symbolic expressions.

As such, we represent these type families as

```
type Loc m :: Nat -> Type
type Value m :: Nat -> Type
type Label m :: Type
```

in Haskell. For example, `Value 8` represents an 8-bit value and `Loc 16` is a pointer to 16 bits.

[1]This is a different concept than is mentioned in Section 2.4.1, though named after the same mathematician.

This key abstraction is extended to QIL's instructions which are all parametric in the number of bits. QIL has no problem allocating a 12-bit location or adding two 4-bit numbers using its generic operations.

For example,

```
alloc  :: AllocMonad m => S n -> m (Loc m n)
add2C  :: (ExecMonad m, KnownNat n) => Value m n
  -> Value m n
  -> m (Value m n)
  -- ... Other semantics
alloc S12  -- has type m (Loc m 12)
add2C (x :: Value m 4) (y  :: Value m 4)  -- has type m (Value m 4)
```

We use Haskell's type-level programming to statically determine operations that may change the size of a `Loc` or `Value`. For instance, when appending bits.

```
appendBits :: (ExecMonad m, KnownNat n, KnownNat n') => Value m n
  -> Value m n'
  -> m (Value m (n + n'))
```

We wrap all these operations in a monad `m`. We require `m` be a monad since Haskell's `do` notation provides convenient sequencing. Specifically, our typeclass `QMonad` provides the associated type families `Loc`, `Value`, `Label`, and `QPtrSize` (the last returning a `Nat`, rather than a type constructor `Nat -> Type`) and a function

```
directBV  :: (QMonad m, KnownNat n) => BitVec n -> m (Value m n)
```

for constructing values. The overall lifter code, which exposes the entire state of a processor (memory locations, RAM, registers, interrupt handlers, and input/output (I/O) ports) and program, is interpreted in an `AllocMonad`. `AllocMonad` extends `QMonad` with functions for allocating memory, defining blocks, etc. The actual code of a block is defined using the `AllocMonad`'s associated execution monad, e.g.

```
newBlock  :: (AllocMonad m, Exec m ()) -> m (Label m)
```

where `Exec` is a type family associated with `AllocMonad`. We require, as part of the definition of `AllocMonad`, that `Exec m` have the same values for the `QMonad` type families as `m` and that `Exec m` implements the `ExecMonad` class.

`ExecMonad` provides the API for local blocks, with all operations for things like writing and reading memory, operating on values, jumping, etc.

At the moment, the final QIL interface is fairly low-level, although already the available set of helper functions far exceeds the number of methods defined as part of classes. We plan, in future releases, to build higher-level APIs for working with these classes, keeping the classes themselves relatively simple so that they can easily be implemented in back-ends.

3.3.1.2 Nominal Encoding. The nominal encoding is "initial" in the sense it encodes QIL via an algebraic data type (dual to the "final" encoding via typeclasses). It is "nominal" in the sense that variables are given names in the data type (represented as integers).

The nominal encoding is much more weakly typed than the final encoding in which type information is included as part of the syntax tree. Haskell's type system is not used to ensure nominal terms are well-typed with respect to QIL's type system (unlike the final case, where only well-typed things are representable). For instance, the type of execution instruction in the nominal encoding begins

```haskell
data ExecIns a where
  ReadLoc :: !ValueV -> !Natural -> !(Location a) -> ExecIns a
  WriteLoc :: !Natural -> !(Location a) -> !Value -> ExecIns a
  AppendBits :: !ValueV
    -> !Natural
    -> !Natural
    -> !Value
    -> !Value
    -> ExecIns a
-- ... Type signatures for remaining operations
```

where we can see we do not parameterize the type `Value` by a size, nor do we ensure that new variables (`ValueV`) have new names.

What is given up in safety here is gained in making it very easy to walk over and manipulate nominal syntax trees–this is especially true because we use `Control.Lens` to provide easy access to `Nominal` objects. For instance, the optimization pass which "de-virtualizes" by replacing dynamic branches with static ones when the value that the dynamic branch would jump to is known requires only six lines of simple Haskell code:

```haskell
getMapping  :: [Binding a] -> Map Integer Label
getMapping = M.fromList . mapMaybe go where
  go (RegBinding l _ i _) = Just (i,l)
  go _ = Nothing

-- Perform the branchKnownVal optimization accross a QIL program
branchKnownVal :: QProgram a -> QProgram a
branchKnownVal p = rewriteOn (allCode.traverse) acc p where
  m = getMapping $ view blocks p
  acc (BranchToValue (LitValue (BV _ v))) = Branch <$> M.lookup v m
  acc e = Nothing
```

We do not expect users to interact with the nominal encoding, except when writing optimization passes. Such passes should be correct, preserving both well-typedness and semantics, and since there is no way to use Haskell's type system to achieve the latter (and that is harder, anyways), it is also up to optimization pass writers to achieve the former.

An implementation of `AllocMonad` is provided which collects a `Nominal` program corresponding to the QIL methods called. Moreover, this translator automatically propagates constants (but does not do constant folding) and replaces the function-based representation of `JoinPoints` and memories with concrete variables.

3.3.1.3 DeBruijn Encoding. Like the nominal encoding, the DeBruijn encoding renders QIL via algebraic data types. Unlike Nominal, this is in the form of *Generalized Algebraic Data Types* where QIL's static type information, including the sizes of values and the return types of instructions, is captured by the type indicies. Only well typed QIL programs are represented in this encoding.

We call this the "DeBruijn" encoding because it uses "De Bruijn indicies" as invented by famed Dutch mathematician Nicolaas Govert de Bruijn and often used for the lambda calculus. Here a variable is represented as an offset, namely, the number of binding before the current location at which that variable was defined. Such a notation is very convenient when we want to combine enforcement of static types with a initial (ADT-based) encoding.

To demonstrate how this works, we will take as our example again the type of execution instructions

```
data ExecIns :: (IOPortInfo -> *) -> Nat -> [Ty] -> [Ty] -> * where
  WriteLoc :: S n
    -> Operand p xs (Location n)
    -> Operand p xs (Value n)
    -> ExecIns p ptrSize xs '[]
  ReadLoc :: S n -> Operand f xs (Location n)
    -> ExecIns f ptrSize xs (One (Value n))
  AppendBits :: S n -> S n'
    -> Operand f xs (Value n)
    -> Operand f xs (Value n')
    -> ExecIns f ptrSize xs (One (Value (n + n')))
-- ... Type signatures for remaining operations
```

here, no variable names are introduced for the new variables, but the QIL types returned by an instruction are captured as the final parameter in its Haskell type. In the line

```
Operand p xs (Location n)
```

we see that the Haskell type (another GADT) of operands is indexed not only by the return type (`Location n`) but also the type of I/O ports `p` and the list of available bound types `xs`. When we string instructions together each will extend `xs` as appropriate.

DeBruijn programs can be universally interpreted in an `AllocMonad`, allowing conversion from `DeBruijn`. Moreover, we provide a function for translating from `Nominal` to `DeBruijn` which dynamically fails (with an error message and in a way which can be handled) if the Nominal QIL program is not well typed.

The primary purpose of DeBruijn is for translation between the other representations, as it breaks up the extremely challenging task of moving from a initial, nominal, untyped encoding to a final, HOAS, typed encoding into two manageable steps.

3.3.2 Data Structure Choices in Encoding.

At present, our Nominal and DeBruijn encodings use linked-list based data structures for sequences of instructions, blocks, etc. This was chosen for ease of implementation, but may be less efficient than other data structures. If Quameleon performance becomes a challenge we will need to reconsider that choice.

Where it was low-cost to do so, we have chosen strict fields for small data, as we expect space savings from avoiding unevaluated thunks.

3.3.3 Overview of Syntactic Elements.

QIL has several fundamental type families which can be referenced by a variable

– Values: these represent bit vectors. In QIL's type system `Values` are parameterized by a natural number of bits.

– Locations: these represent assignable locations where values can be read and written. In QIL's type system `Locations` are parameterized by a natural number which denotes the size of `Values` storable there.

– Labels: these represent program locations and can be jumped to.

– RamSelectors: these represent families of `Locations` indexed by `Values`. RamSelectors may be needed to distinguish between multiple types of memory, for example, a page table compared to physical memory.

– Ports: these allow treating a Location (including memory regions) as I/O. Some languages refer to these as *volatile*, meaning they may be read from or written to externally and asynchronously.

– JoinPoints: these represent a local continuation that can be jumped to. Unlike a `Label`, which denotes a global location, a `JoinPoints` type is parameterized by a list of argument types.

From these, we form instructions, blocks, and programs. A QIL program consists of four pieces of information:

1. a globally defined code-pointer size (a natural number)
2. a sequence of allocation instructions defining registers and memories
3. a sequence of blocks, each binding a label
4. optionally, a label called the “entry point.”

Blocks bind labels one of two ways. Registered blocks can be used for static jumps (with an associated label) or dynamic jumps (with an associated address). Unregistered blocks only have a label.

Generally, in a lifter, a registered block is (at least initially) generated for each instruction. Optimization may then generate additional blocks or combine blocks via inlining.

In either kind of block, the body of the block consists of a sequence of instructions which may bind variables. Each variable is bound exactly once in the style of *static single assignment* (SSA). Unlike many SSA ILs, there are no “labels” or “ϕ nodes” inside a QIL block. Instead, block-local control is achieved by way of structured control flow consisting of:

– Boolean (if) and multiway (case) conditional statements

– let-bound join points (which take parameters such as functions in high-level languages).

Listing 3.1 A fragment of M6800 assembly.

```
LDA A #15   ; A <- 0xF
AND A $40   ; A <- A & [0x40]
```

3.3.3.1 Other Representations. We also support JSON output for our angr bridge and a pretty-printed, human-readable syntax as shown in Listing 3.2. At present QIL does *not* support non-sequential semantics and self-modifying programs. We discuss these limitations in the future work section.

3.3.4 Overview of Semantics. A QIL program takes as its denotation a labeled transition system where the labels on transitions are sequences of well-typed reads and writes to some set of locations.

As is standard, we think of the abstract state of the program's denotation as coming from two parts: program locations (that is, the QIL labels bound by blocks) and the other state, the latter of which is described by the variables (RAM, registers, etc.) defined in the allocation section of the QIL program.

By computing the denotation of each block body, we can easily compute the denotation of the entire program. Specifically, we start with, as states, the Cartesian product of the set of labels and the denotation of the domain of memory (both RAM and registers), and then for every element of the denotation of a block we add the appropriate transitions, adding intermediate steps as necessary.

Note that, crucially, while QIL blocks can be non-deterministic, they must terminate. No fancy denotational techniques are needed to account for non-termination, as it exists only at the top-level of the semantics (the transition system).

3.3.5 A Worked Example Program. In order to demonstrate the QIL language, consider the fragment of M6800 assembly language in Listing 3.1 which takes the bitwise "and" of `0xF` and the byte at location `0x40`.

The pretty-printed QIL is given in Listing 3.2. Note that this program is unoptimized: after sufficient optimization this program would reduce to a precomputed write since the program has no inputs.

Line 1 indicates this program uses 16-bit values for dynamic jumps. Lines 2–8 set up the memory used by this machine. For instance, Line 3 creates an 8-bit assignable memory location

Listing 3.2 A worked QIL example.

```
code_ptr_size: S16
alloc_part: {
   &1 := alloc[S8] // Register A
   &2 := alloc[S8] // Register B
   // ... Other registers
   &6 := alloc[S1] // Carry Flag
   &7 := alloc[S1] // Overflow Flag
   // ... Other status flags
   MEM(1) := buildMemory[S16 S8]
}
code_part: {
   @1 := block { }
   @2 := registered_block "AND A (DIR8 64)" 2 {
      %1  := readLoc[S8] &1         // read Register A
      &12 := MEM(1)[S16].BV[S8](40) // Memory location 0x40
      %2  := readLoc[S8] &12
      %3  := AndBit[S8] %1 %2
      writeLoc[S8] &1 %3            // set Register A
      branch @1
   }
   @3 := registered_block "LDA A (IMM8 15)" 0 {
      writeLoc[S8] &1 BV[S8](f)     // set Register A
      branch @2
   }
   @4 := block {
      branch @3
   }
}
entry_point: @4
```

(i.e. a register) and associates it with the name `&1`. In the QIL syntax all location variables start with an ampersand and the comment `Register A` is just metadata.

Our M6800 lifter ends up generating blocks in the opposite order from the instructions. As such, the initial instruction, set as the entry point, has the label `@4`. Next, the program branches to the label `@3`. Line 20 states the location `&1` gets the value 15 (0xF), and its size is 8 bits. The other potentially confusing line is 13, which reads 8 bits from the 16-bit memory location 64 (hex 0x40).

3.4 Optimizations

Quameleon provides several optimization passes with the goal of decreasing code size and increasing analyzability. One example is constant folding, which consists of replacing a variable with its value when that value can be statically known. Other optimizations we have implemented include unreachable code elimination and inlining.

3.5 Analysis Back-Ends

We have currently implemented two analysis back-ends.

1. A bridge between Quameleon and angr. This allows loading QIL programs so that QIL appears as simply another binary format. We include metadata such as the register names inside this JSON to provide similar convenience to existing ISAs.

2. Concrete execution. This back-end provides the ability to interpret QIL programs; i.e. an emulator for supported architectures. Our general purpose interpreter takes a set of call-backs for resolving I/O effects, early termination, or non-deterministic calls. As such, we provide a unified back-end for both pure and side-effectful interpretation strategies.

3.6 Case Study: Symbolic Execution

One classic use-case of binary analyses is to check for vulnerabilities in an executable, such as a back door in a password checking function. We present a highly-simplified version of such a program in Listing 3.3.

We show an example analysis case for a simple program and how to analyze its corresponding binary. This example works under the assumption that a lifter has been implemented for a new ISA, and I/O works by calling a special assembly instruction `iop` in either reading or writing mode, which then fills or reads from a register accordingly. Under QIL terminology, these I/O operations read and write from *Ports*. The code generated by `putc` and

Listing 3.3 Example C code with a very simple password, output in QIL as `charpwd.json`.

```
inline void putc(char c);
inline unsigned char getc(void);
int main() {
    char c;
    putc('h');
    putc('i');
    putc('\n');
    c = getc();
    if (c == 'p')
        return 0;
    else
        return 1;
}
inline void putc(char c) {
    asm volatile ("iop w d%0"
                  :
                  : "d"(c));
}
inline unsigned char getc(void) {
    unsigned int o;
    asm volatile ("iop r d%0"
                  : "=d"(o));
    return o;
}
```

`getc` is irrelevant as long as the reads and writes are tracked internally by QIL: some assembly languages may use memory-mapped I/O or special assembly instructions.

Our goal is to show the two possible states of this program and which input correspond to each state. While the C code looks almost trivial, the behavior of a binary is much more obfuscated.

We accomplish this by first emitting the decompiled QIL code as JSON output. The example in Listing 3.3 becomes `charpwd.json`, to be read into angr using a script whose simplified version is show in Listing 3.4. This allows us to leverage the power of angr since we only need to write one interpreter for angr. That is, angr treats QIL as a binary, and we may write our semantics using the expressiveness and type-safety of Haskell.

On the notion of equality: a given program behavior is defined by its sequence of inputs and outputs. This allows, for example, optimizations to be still considered identical behavior.

Additionally, given a set of variables (represented as bitvectors), angr can be used to exhaustively check the output of a state space using an SMT solver such as Z3 searching for either

Listing 3.4 Using angr with the QIL back-end.

```python
# Some snippets for interactive angr-qil binary analysis
from angr_platforms.qil import *
import angr
p = angr.Project('charpwd.json')
ms = p.factory.simgr()
s = p.factory.entry_state()
for i in range(1000):
    print("Step {}\t Ins: {}".format(i,
        s.qil.instruction if hasattr(s.qil, "instruction") else "??"))
    print("\tReg1: {}\n\tReg2: {}".format(s.regs.Reg1, s.regs.Reg2)
    print("\treads:{}".format(s.ports.ConsoleIO.dump_reads()))
    print("\twrite:{}".format(s.ports.ConsoleIO.dump_writes()))
    succ = s.step()
    if len(succ.successors) > 1:
        break
    s = succ.successors[0]

# Print where the split occurs
for (idx,n) in enumerate(succ.successors):
    print("State: {}".format(idx))
    print("\t{}".format(n.ports.ConsoleIO.dump_reads()))
    print("\t{}".format(n.ports.ConsoleIO.dump_writes()))
```

Listing 3.5 The tool angr correctly detects two possible outcomes, State 1 indicating the password “p” was provided as input on Line 5.

```
1 State 0:
2      b'\x00\x00'
3      b'\x00h\x00i\xff\xfa\x00\x00\xff\xfa'
4 State 1:
5      b'\x00\xp'
6      b'\x00h\x00i\xff\xfa\x00p\xff\xfa'
```

satisfiability with constraints or to enumerate all reachable states. Applying symbolic execution using angr as shown in Listing 3.4 permits us to analyze the binary to figure out the password of our simple program in Listing 3.3. I/O reads are printed according to Line 21 in Listing 3.3 and the output is shown in Listing 3.5.

One issue which is not indicated by such a simple example is the execution time. SMT solvers or exhaustive state space enumeration in the worst case have exponential runtime in the number of instructions of a binary; so fully-automated analysis typically does not work for large binaries. To overcome this, in practice, angr is used semi-interactively to investigate the most common search spaces. Our use case is to be completely exhaustive, and so to get more scalable anaysis of QIL programs we must leverage the QIL-QIL optimizations.

3.7 Related Work

Related work includes analysis tools such as BAP (Binary Analysis Platform) [33], B2R2 [106], and angr [176]. In particular, angr has a large user community and a substantial degree of completeness. Unfortunately, neither angr nor BAP supported the ISAs we needed. We note the differing design goals of other tools to motivate the overall design of Quameleon.

– Our goal is to generalize both front-ends and back-ends for binaries which we know at some level the expected behavior, but require high assurance of correctness; this contrasts with angr's design goals of being primarily for reverse engineering adversarial binaries using symbolic execution and heuristics.

– Both angr and BAP use ILs based on mutable temporaries by default. Instead, we wanted a static single assignment (SSA)-based IL. LLVM [121] uses SSA, but is aimed for optimization rather than binary analysis.

Additionally, SAIL [5] and K [162] are domain-specific languages used to specify ISA semantics which have a similar goal of generating emulators and analysis back-ends from ISA specifications.

3.8 Conclusion and Future Work

We presented Quameleon, a tool for sound binary analysis designed from the beginning to be easily extended to different architectures and types of analysis. We accomplish this by lifting our input ISAs into an intermediate language QIL, an SSA-based intermediate language. QIL programs are amenable to analysis because they make explicit all effects of an assembly language program and the small core language facilitates our effort to formalize QIL in a proof assistant such as Coq.

The first desired feature would be to support self-modifying binaries. Our idea to this end is to extend QIL with an (optional) special block handling branching to values not known until runtime, wherein QIL could look up the location in memory, decode its contents to an instruction, then evaluate that instruction. We also wish to add additional back-ends as listed in Figure 6 such as Hoare Logic-style predicate transformers and abstract interpretation. Lastly, QIL does not include floating-point instructions unlike many other ILs. We are exploring what it would take to develop a formal theory of all floating-point representations to be used in QIL that would be generic enough for pre-IEEE 754 floating-point formats.

The next chapters of this dissertation give detailed static analyses of different floating-point operations. One exciting future direction would be to permit analysis of floating-point programs using the statistical methods of Chapter IV as well as the sound error analysis of Chapter V included in Quameleon. This would expand the notion of correct behavior of floating-point programs in Quameleon from exact bit-level behavior into one based on more realistic real-number semantics.

Another result of extensibility being a primary design goal is the ability to extend to old ISAs; languages with non-byte addressable memory, pre-IEEE 754 floating-point arithmetic or requiring cycle-accurate emulation can all be added without modification QIL's core architecture.

Quameleon allows us to apply sophisticated programming language features of Haskell's typechecker to ensure binaries and their lifted representations are well-typed across all intermediate representations. This helps with assurance by providing strong static guarantees of program behavior, such as ensuring integer overflows exception handling behave precisely according to the specification of the original architectures. These assurances are preserved across the various internal representations of QIL and with a bridge to the symbolic execution framework of angr, users can statically analyze programs to verify their behavior across entire input spaces.

In the beginning of this dissertation we mentioned two places where low-level programming is done: situations where it is required, such as bootloaders or compilers, and situations where only hand-written assembly can achieve sufficient performance. Quameleon can technically be used in both places. However, much of performance-critical code in use consists of mainly floating-point operations (FLOP) on highly parallel machines. To wit, the M6800 ISA does not even have a floating-point unit. The remainder of this dissertation focuses on floating-point arithmetic. We begin by analyzing one aspect of the Message Passing Interface (MPI) standard to better understand its numerical properties.

CHAPTER IV

A STATISTICAL ANALYSIS OF ERROR IN MPI REDUCTION OPERATIONS

Nota Bene: This work is based on previously published co-authored work [152]. Samuel D. Pollard was the main author and implemented all the source code and wrote most of the paper. Boyana Norris was the principal investigator for this research and provided research direction, writing, and proofreading of the paper.

4.1 Introduction

Message Passing Interface (MPI) [43] is the industry standard for distributed, high-performance computing (HPC) applications. With MPI, collective operations assume an associative operator; however, floating-point (FP) arithmetic is in general nonassociative. This creates a predicament: on one hand, the nondeterminism of parallel operations is a cornerstone of the scalability of parallel computations. On the other hand, bitwise reproducibility is much more difficult to preserve if nondeterministic operation ordering is permitted with floating-point operations (FLOPs). To make matters worse, some small discrepancies are acceptable in many computational models, such as iterative solvers for linear systems [181]. Thus, the need for rigorous analysis of nonassociativity in HPC applications is twofold: both to ensure reproducibility where needed and to determine when these errors are inconsequential.

With the flexibility of the MPI standard, results are not guaranteed to be the same across different network topologies or even across different numbers of processes [189]. But we must not paint too pessimistic of a picture. In reality, the hard work of MPI developers has resulted in overall stable algorithms, and popular implementations such as OpenMPI and MPICH evaluate to the same result when the function is applied to the same arguments appearing in the same order [9]. However, this order is not necessarily *canonical* and so may differ from the serial semantics developers might expect.

4.1.1 Hypothesis.

Our original hypothesis for this chapter was the following: selecting between different reduction schemes and reduction algorithms will cause a significant change in accuracy. Our first case studies indicate the opposite—reduction algorithms have little or no effect on the final result. Instead, the dynamic range of the summands is much more significant. We predict this is true because MPI reduction algorithms by nature strive to create a balanced reduction tree for efficiency. As formulated by Espelid [69], summation error (and in turn, MPI reduction error) consists of two independent parts: the initial error (based on the range of the inputs) and the algorithm error. Though there exist pathological orderings in any case to give huge algorithm errors, the optimal way to minimize summation error is to minimize the magnitudes of intermediate sums [69]. For many inputs, this goal is supplementary to creating balanced reduction trees.

4.1.2 Contributions. Our work analyzes error of MPI reduction algorithms several different ways. In this work, we:

– generate samples from every possible reduction strategy permitted by the MPI standard (Section 4.2);

– calculate previously-established analytical bounds and probabilistic estimators as they apply to some sample input probability distributions (Section 4.3);

– show experimental results for different probability distributions and summation algorithms as they relate to the MPI standard (Section 4.4);

– investigate the correlation between reduction tree height and summation error (Section 4.4.3); and

– consider the effect allreduce algorithms have on the accuracy of the proxy application Nekbone (Section 4.5).

4.2 State Space of MPI Reduction Operations

The goal of our work is to explore the implications of assuming associativity of FLOPs with respect to MPI reduction operations. We first explain the concepts behind MPI reduction algorithms, then describe the state space of different ways these reductions can be performed.

4.2.1 Reduction Operations. Conceptually, an MPI reduction operation works by repeatedly applying a binary operation to a given input, thereby reducing its size and returning an accumulated value. More concretely, one use of `MPI_Reduce` is to compute the sum of all elements of an array in parallel, and transmit the sum to a single process.

A reduction operation is typically parameterized by its associativity. For a one-dimensional array, the natural choices are left or right-associative. Imperative, serial implementations typically imply left-associativity since that matches C and Fortran specifications

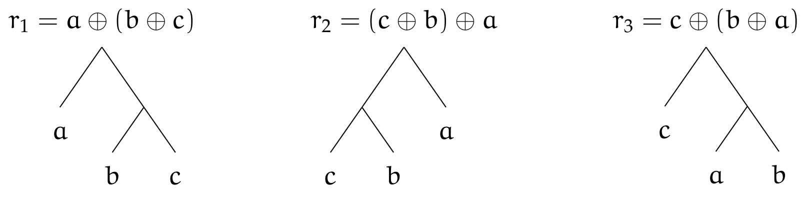

$r_1 = a \oplus (b \oplus c)$
a
b
c
$r_2 = (c \oplus b) \oplus a$
c
b
a
$r_3 = c \oplus (b \oplus a)$
c
a
b

Figure 7. With the commutative but nonassociative operator $\oplus$, $r_1 = r_2$ but $r_2 \neq r_3$.

for + and * and is strongly idiomatic. Recall that FP addition and multiplication are commutative but nonassociative operations. With MPI there is no fixed associativity, so many results are permitted for an operation such as `MPI_Reduce`.

This paper focuses on FP addition. By assuming associativity and commutativity, MPI's definition of a reduction operation permits many different ways to reduce an array of values. For example, Figure 7 gives three different ways to reduce a three-element array. Each reduction strategy forms a *reduction tree*.

Previous work used randomly shuffled arrays to check FP summation error empirically [38]. We expand this method by also randomizing the reduction tree. While selecting summation order and associativity is not a new idea [94], we provide more results as they specifically relate to MPI. The state space we describe is all possible reduction trees and could also be applied to other parallel programming paradigms. Next, we analyze the possible state space of how collective operations can be performed.

4.2.2 Quantifying the State Space of Possible Reductions. The MPI 3.1 standard [136] has the following description:

> The operation `op` is always assumed to be associative. All predefined operations are also assumed to be commutative. . . However, the implementation can take advantage of associativity, or associativity and commutativity, in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating-point addition.

For custom operations, the issue is similar:

> If `commute = false`, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, taking advantage of the associativity of the operation. If `commute =`

`true` then the order of evaluation can be changed, taking advantage of commutativity and associativity.

A *rank* in MPI terminology refers to one MPI process. Characterizing these two descriptions gives us two different families of possible results for a reduction operation. We compare these two families with the most straightforward serial implementation as well as previous work.

Throughout, we assume a one-dimensional array A of FP numbers of length n. We use A_k to denote array access with $k \in \{1, 2, \ldots, n\}$. Henceforth, we refer to the combination of permuting A along with a particular reduction tree as a *reduction strategy* of A.

We point out a slight clarification of our use of *random* throughout this paper. In reality, in reduction algorithms the order or associativity is not *random* but *unspecified*. We say *random* simply for notational consistency; in Section 4.4 we generate random reduction strategies according to these families. Now we present four families of reduction algorithms: two described by the MPI standard and two others.

1. Canonical. This is left-associative and defines a single possible reduction order.

2. FORA (**F**ixed **O**rder, **R**andom **A**ssociation). This represents the case when `commute = false`. The operations may be parenthesized, but the order is unchanged. For example, in Figure 7, r_1 is an acceptable sum but r_2 and r_3 are not. This is a well-known combinatorial problem and there are C_n possible ways to parenthesize n values, where
$$C_n = \frac{(2n)!}{(n+1)!n!}$$
is the nth Catalan number [113, §7.2]. Each parenthesization corresponds to a binary tree, but for convenience we call these *associations*.

3. RORA (**R**andom **O**rder, **R**andom **A**ssociation). This is the default behavior when using `MPI_Reduce`. Operations may be reordered assuming both commutativity and associativity. Counting the possible number of different equivalent reduction trees is a less well-known, but still solved, combinatorial problem [53]. At first glance, this would appear to have C_n different reductions for each of the $n!$ possible permutations of A. However, because of commutativity, many of these reduce to the same answer. Let g_n be the number of possible

different reduction trees. As it turns out,

$$g_n = (2n - 3)!! \tag{4.1}$$

where !! is the double factorial (in this case on odd integers). That is, $(2n-3)!! = 1 \times 3 \times 5 \times \ldots \times 2n - 3$.

4. ROLA (**R**andom **O**rder, **L**eft **A**ssociative). This is not precisely described as part of the MPI standard but is used in previous work [38] and could still arise for algorithms where the array's order is nondeterministic, but reduction is still done in canonical order. Here, the array may be shuffled but the reduction is left-associative. There are $n!/2$ different possible orderings. To see this, note there are $n!$ permutations, and by commutativity only the first two elements can be rearranged; the rest are fixed by left-associativity.

For the last three cases the growth rate of the number of possible reduction orders is exponential; exhaustively checking results is infeasible which is why we generate reduction strategies randomly. Comparing the last three cases, we have

$$C_n < \frac{n!}{2} < g_n \tag{4.2}$$

for $n > 4$. The proof of this is given in Appendix A.

We discuss further these four cases and how they relate to the MPI standard. There are g_n distinct values for ROLA and $n!/2$ for RORA. Since $n!/2 < g_n$, there must be at least one reduction strategy that occurs in ROLA but not in RORA (this argument is referred to as the *pigeonhole principle*). Therefore, ROLA cannot generate every possible value permitted in RORA. For example, $(bd)(ac)$ can be constructed via RORA but not via ROLA because we cannot convert every left-associative reduction tree into a balanced binary tree without associativity.

To reiterate, case 1) is canonical, case 2) (FORA) is precisely the possible state space for custom MPI operations when `commute = false`, and case 3) (RORA) is the state space for the default MPI case. For comparison to previous work, we include the case 4) (ROLA) in the subsequent analysis.

4.2.3 Generating Samples. We begin by generating random binary trees—because every possible association of a sequence of nonassociative binary operations can be described by a binary tree. Thus, when performing a reduction on n elements, we wish to sample from the space of every binary tree with n leaves. To do this we use Algorithm R (Rémy's procedure) [113].

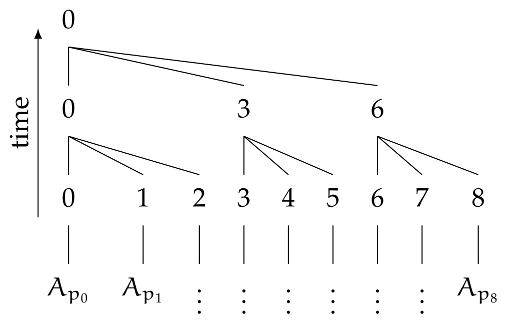


Figure 8. One possible reduction algorithm. The array A is split up across 9 MPI ranks. Each p_i indicates contiguous chunks of A. The final result is obtained by sending partial sums "up" the tree.

For the FORA case, we generate a random binary tree. For the RORA case, we first shuffle the array, then generate a random binary tree by which to sum the array. For the ROLA case, we shuffle the array, then accumulate in a left-associative manner. Some examples of random reduction trees are given in Figure 7.

4.2.4 The Nondeterminism of Reduction Algorithms.

We describe the entire state space of possible reduction strategies, but realistically, not every reduction strategy will occur for a given topology and reduction algorithm. For more sophisticated reduction algorithms such as Rabenseifner's algorithm [185], it is not clear how to limit the state space.

In addition, it is not immediately obvious where nondeterminism, given a particular algorithm, can arise. We show an example in Figure 8 for visualization. In this case, nondeterminism can arise either from messages arriving in different order for the intermediate sums, or the array being distributed differently according to topology. Depending on the internal algorithm of the reduction, one may then be in any of the four cases. Another place for nondeterminism is in how the array is distributed across nodes. For topology-aware reduction algorithms, it may be the case that nodes are not in canonical order in the reduction tree, which is permitted by the MPI standard.

Finally, notice that *within* a rank, the values will probably be reduced in the most straightforward way—left-associative. We do not assume this, but can easily model it by using an array whose size is equal to the number of nodes and whose elements are the partial sums within a rank.

4.2.5 Generating Random Floating-Point Values. In most cases, we use the Marsaglia-Multicarry [133] pseudorandom number generator to generate uniform distributions with a fixed seed across experiments. We denote a uniform distribution over the half-open interval $[a, b)$ as $U(a, b)$.

Uniform random number generators for FP numbers typically generate an integer in some range $[0, K)$, then divide by K to get a value in $[0, 1)$. However, if $K < 2^{1022}$ then it is impossible to generate every FP number in $[0, 1)$. Notably, no subnormal numbers (for double-precision, numbers in $[0, 2^{-1022})$) are generated. We alleviate this by generating a special distribution which we call *subn*. The process for generating these is simple: generate 62 random bits, then clear the highest two. This ensures the exponent is in the range $[-1023, 0]$ and so every FP number in $[0, 2)$ can be generated. Sampling from this distribution gives an expected value of 0.00282 and looks roughly exponentially distributed, with rate parameter $\lambda \approx 1/0.00282 \approx 354$.

4.3 Analytical Error Bounds and Estimators

Analytical error of floating-point summation in the general case can be pessimistic. We wish to see how pessimistic. First, we introduce some helpful notation. Let ε be machine epsilon, or the upper bound on relative rounding error from a FLOP. That is, for some real operator op and its corresponding FP operator $\odot$,

$$x \text{ op } y = (x \odot y)(1 + \delta) \text{ and } |\delta| \leqslant \varepsilon. \tag{4.3}$$

For our experiments, we use IEEE 754 *binary64* format so $\varepsilon = 2^{-53}$. We focus on FP addition, notated as $\oplus$ to distinguish from real number addition. We put $\oplus$ above the summation symbol to denote FP sum with an unspecified reduction strategy like so: $\sum^{\oplus}$.

One last bit of notation: let S_A be the true value of the sum of every element of A. A well established result by Wilkinson [202] bounds the absolute error of FP summation as:

$$\left| \sum_{k=1}^{\oplus\, n} A_k - S_A \right| \leqslant \varepsilon(n-1) \sum_{k=1}^{n} |A_k| + O(\varepsilon^2). \tag{4.4}$$

This bound can be refined with some assumptions. Existing statistical analysis by Robertazzi and Schwartz [159] derives expected estimators of the relative error of FP summation in special cases. If you assume

1. The numbers are positive and either uniformly or exponentially distributed (for all k, $A_k \sim U(0, 2\mu)$ or $\exp(1/\mu)$);

2. floating-point addition errors are independent, distributed with mean 0 and variance σ_e^2; and

3. the summation ordering is random;

then the relative error is approximated by

$$\frac{1}{3}\mu^2 n^3 \sigma_e^2. \tag{4.5}$$

Assuming (4.3) holds and summation error is uniformly distributed in $[-\varepsilon/2, \varepsilon/2]$ yields

$$\sigma_e^2 = \frac{1}{12}\varepsilon^2. \tag{4.6}$$

These equations are not particularly useful in isolation, but we use this estimator to verify our empirical results in the following section.

4.4 Empirical Results

We compare the reduction strategies in Section 4.2 for reduction along with some baselines. We list how each is generated here:

i) Canonical (left-associative). The code is essentially `s=0.0; for(k=0;k<n;k++) s+=A[k];` for IEEE 754 double-precision.

ii) FORA (**F**ixed **O**rder, **R**andom **A**ssociation). Generates case 2) reduction strategies. This is done by generating a random reduction tree but leaving the array in its original order.

iii) RORA (**R**andom **O**rder, **R**andom **A**ssociation). Generates case 3). This is done by shuffling the array, then generating a random reduction tree and reducing over that.

iv) ROLA (**R**andom **O**rder, **L**eft **A**ssociative). This is done by shuffling the array, then reducing left-associative. This is for comparison with previous work [38].

v) MPFR. We use the Multiple Precision Floating-Point Reliable (MPFR) library by taking the randomly-generated double-precision numbers as exact then convert to 1,000-digit (3,324-bit) MPFR floats [76] and reduce using MPFR's correctly-rounded operations in left-associative order. We use this as the ground truth.

Using 3,324 bits for MPFR may seem excessive, but a proposal by Kulisch describes a 4,288-bit accumulator to compute exact dot products for double-precision floats [114].

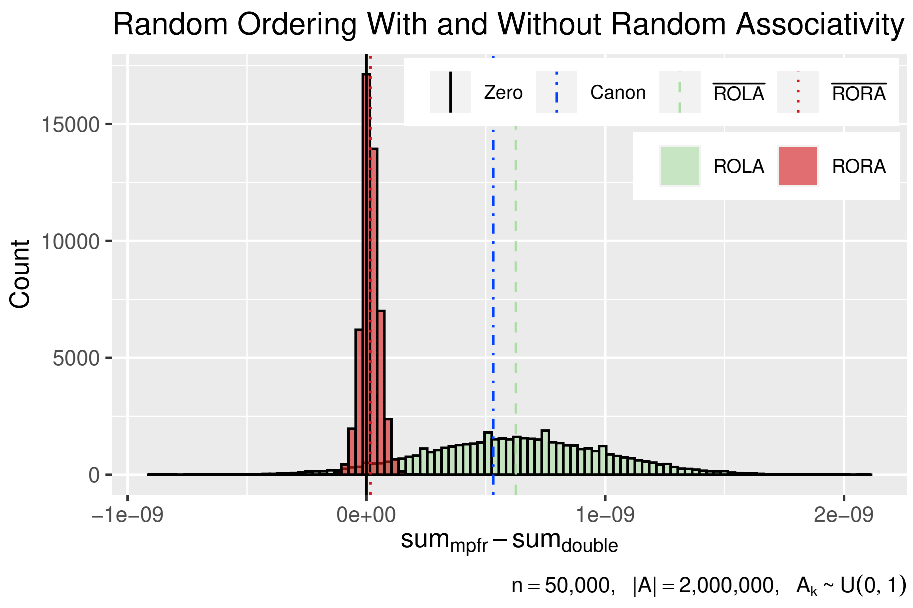


Figure 9. On average, left-associative summation results in a larger error than random associativity. Left-associative, in MPI terms, would be a canonical ordering. An $\overline{\text{overline}}$ indicates the arithmetic mean of all samples.

Multiplication requires more bits to account for the product of very large and very near-zero numbers, so for summations 3,324 digits are sufficient.

4.4.1 Distribution of Summation Errors. In Figures 9, 10, and 11 we present a set of histograms showing the error of different MPI reduction schemes. All of these are histograms with a sample size of 50,000, but the x-axes are different measurements of error.

In Figure 9, we compare not the absolute or relative error, but simply the *difference* between the true value (sum_{mpfr}) and the computed value ($\text{sum}_{\text{double}}$). We see here that random ordering *and* random association, on average, provides a far tighter error bound compared to enforcing left-associativity. Not only that, but enforcing left-associativity on a shuffled array (ROLA) underestimates the true sum. Higham and Mary show this is expected: as the partial sums get sufficiently large, adding a small positive number will not change the sum; notationally, for sufficiently large S and small x, $S \oplus x = S$ [97]. Other distributions have similar behavior, where ROLA gives a more biased sum compared to RORA. This provides evidence that MPI reduction error may not be as severe as indicated in the work of Chapp et al [38]. For tabulation of the vertical lines in Figure 9, see Table 6.

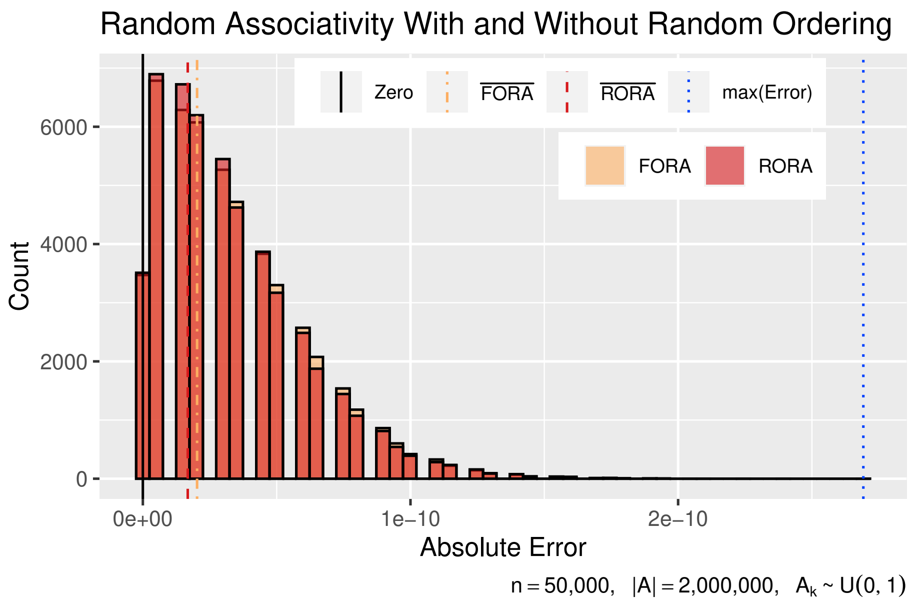


Figure 10. Here we see shuffling the array has little effect on the summation error. The number of bins was chosen to be exactly the number of unique different floating-point numbers. The apparent missing numbers is *not* an artifact of plotting but rather floating-point numbers which occurred infrequently (or just once).

Looking at Figure 9, we can also compare our refinement of different FP summation. The similarity of ROLA (case iv, used by Chapp et al.) and RORA (case iii, everything permitted by the MPI standard). The smaller error of RORA indicates MPI summation can be quite robust if the inputs are randomly distributed similar to our approach.

We compare the distributions for FORA and RORA separately in Figure 10 because they look similar. This indicates that the order in which elements are added has a smaller effect compared to the associativity. This makes sense in the context of our random number generation: previous work indicates summation error is dependent on the magnitudes of the intermediate sums [202]. Randomly shuffling the array would not affect the magnitude of intermediate sums much since the summands are independently and identically distributed (iid). Further research is needed to determine whether this holds for input which is not iid.

We point out some interesting things about Figure 11. For one, the greatest relative error among everything we sampled is with the $U(-1, 1)$ distribution. One possible explanation is that numbers close together have a higher chance of having large cancellation; this is the classic

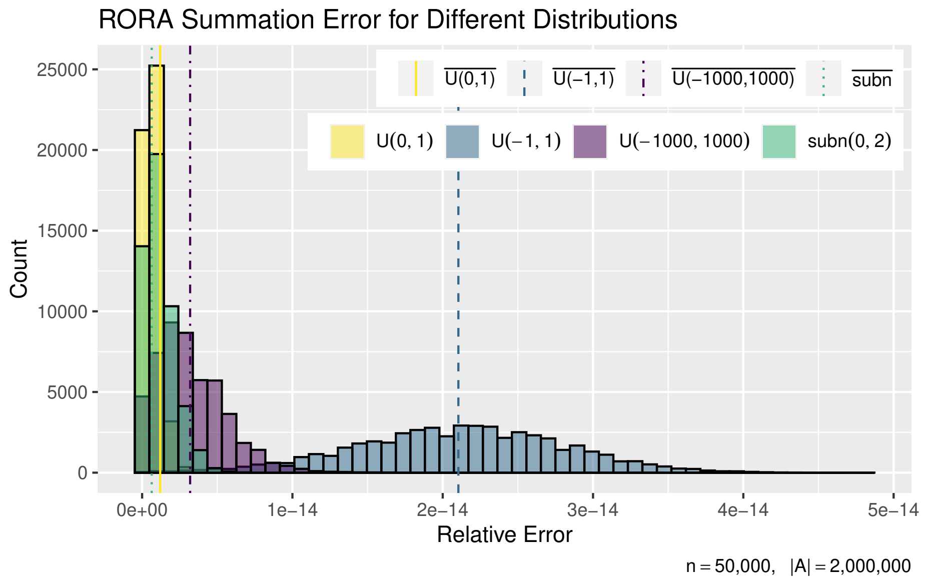


Figure 11. We notice a wide variation of errors between different distributions with $U(-1, 1)$ having the largest average relative error. We measure relative error as $|\text{sum}_{\text{double}} - \text{sum}_{\text{mpfr}}|/\text{sum}_{\text{mpfr}}$.

example of floating-point nonassociativity. That is, if $x \approx y$ then $x \ominus y$ is close to zero. Then, if for another intermediate summand $z \gg (x \ominus y)$ more error is exposed when computing $z \oplus (x \ominus y)$.

We tabulate some important values (displayed as vertical lines in Figs. 10, 11, and 9) alongside their analytical relative error bounds in Table 6.

4.4.2 Comparing Empirical and Analytical Bounds. We calculate the "Robertazzi" row in Table 6 by substituting our experimental values into (4.5) and (4.6). Hence, for $U(0, 1)$ and $n = 2 \times 10^6$,

$$\frac{1}{3}(0.5)^2(2 \times 10^6)^3 \left(\frac{1}{12}\varepsilon^2\right) \approx 6.848 \times 10^{-16}.$$

This estimator is very close to our observed results (See $U(0, 1)$ $\overline{\text{RORA}}$ in Table 6), but Robertazzi's work only applies for summation of either uniform or exponentially distributed positive numbers.

For subn, we get an impossibly small expected relative error since it is much smaller than machine ε. This is to be expected, however, since subnormal numbers lack the same precision as normal numbers, and serves as more evidence that FP summation can go awry for numbers very close to zero (in the case of double-precision, numbers smaller than 2^{-1022}).

Table 6. Table of significant values for our empirical results. Smaller is better.

Distribution	Measurement	Relative Error	Note
$U(-1,1)$	$\overline{\text{RORA}}$	2.104×10^{-14}	
$U(-1,1)$	max(RORA)	4.824×10^{-14}	
$U(-1,1)$	$\overline{\text{ROLA}}$	8.358×10^{-12}	max. empirical
$U(-1,1)$	Canonical	6.124×10^{-12}	
$U(-1,1)$	Analytical	7.951×10^{-7}	
$U(0,1)$	$\overline{\text{RORA}}$	6.702×10^{-16}	
$U(0,1)$	max(RORA)	4.073×10^{-15}	
$U(0,1)$	$\overline{\text{ROLA}}$	1.282×10^{-14}	
$U(0,1)$	Canonical	1.062×10^{-14}	
$U(0,1)$	Analytical	1.776×10^{-8}	
$U(0,1)$	Robertazzi	6.848×10^{-16}	
$U(-1000,1000)$	$\overline{\text{RORA}}$	3.194×10^{-15}	
$U(-1000,1000)$	max(RORA)	2.084×10^{-14}	
$U(-1000,1000)$	$\overline{\text{ROLA}}$	3.711×10^{-14}	
$U(-1000,1000)$	Canonical	7.503×10^{-15}	
$U(-1000,1000)$	Analytical	7.951×10^{-7}	max. analytical
subn	$\overline{\text{RORA}}$	6.383×10^{-16}	min. empirical
subn	max(RORA)	4.385×10^{-15}	
subn	Canonical	6.945×10^{-14}	
subn	Analytical	1.713×10^{-7}	min. analytical
subn	Robertazzi	4.206×10^{-24}	not achievable
	machine ε	1.110×10^{-16}	double-precision

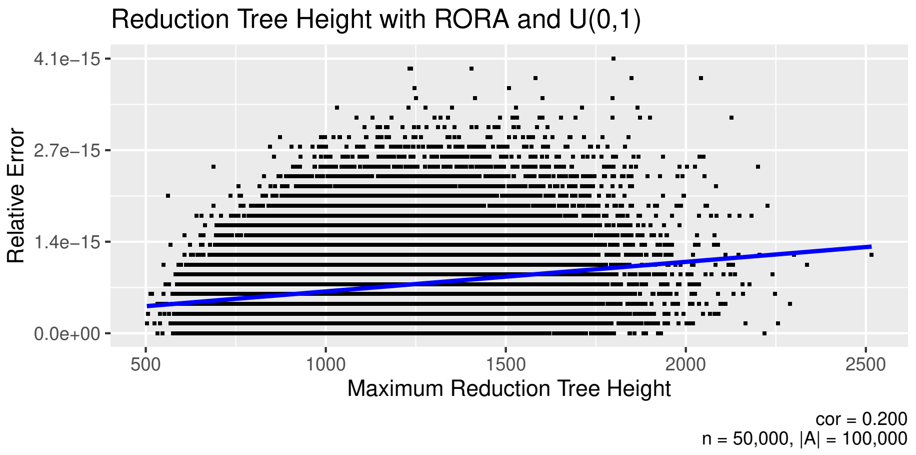


Figure 12. The bands are a result of the gap between floating-point numbers.

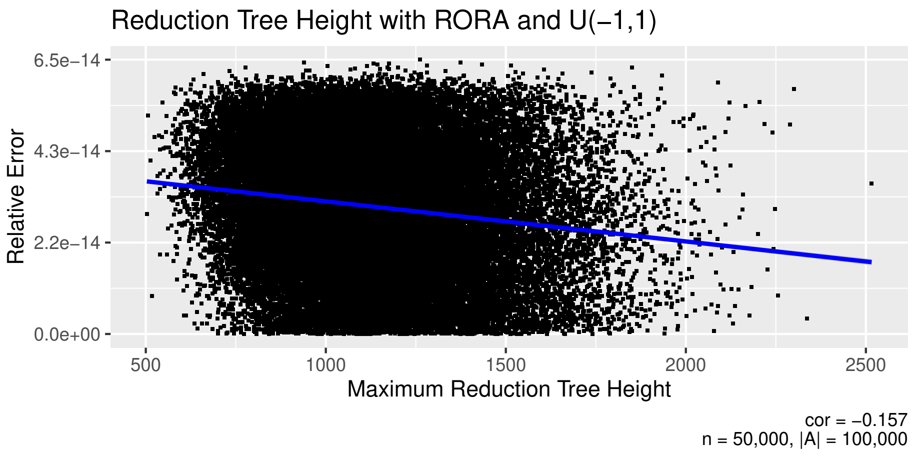


Figure 13. The $U(-1, 1)$ distribution has a larger error than $U(0, 1)$.

For the array size of our experiments (2 million elements), we see the analytical bounds being particularly pessimistic: they are between 6 and 7 orders of magnitude *larger* than our observed results.

4.4.3 Reduction Tree Height and Error. When trying to answer the question of how to make our reduction strategy (recall this is the combination of array ordering and reduction tree shape), we ask: does reduction tree height affect summation error? We looked at the maximum height of each reduction tree for our experiments. Figure 12 shows a very weak correlation, with correlation coefficient $\rho = 0.2$. We see a *negative* correlation with the $U(-1, 1)$ distribution in Figure 13. However, we still plot these for completeness' sake.

Now that we have seen some empirical examples from manufactured arrays, we compare our results with a more realistic use-case.

4.5 Nekbone and SimGrid: A (Robust) Case Study

SimGrid is an ambitious project to simulate MPI applications [37]. We use two of the many features of SimGrid in this work: its simulation of network topology and its implementation of a large number of MPI reduction algorithms across different MPI implementations including MPICH, OpenMPI, Intel MPI, and MVAPICH[190]. Table 7 describes each of these algorithms. We do not use every allreduce algorithm that SimGrid implements because some do not work for processor counts which are not a power of two and our simulated topology is a 72-node fat tree cluster.

Nekbone is a proxy application for the Nek5000 physics simulation code [73]. Nek5000 is a powerful HPC application for solving various computational fluid dynamics problems [74]. Nekbone is a simplified version of Nek5000 containing only some key computational elements by solving a three-dimensional Poisson equation using the conjugate gradient iterative solver. We run Nekbone's default settings: 101 iterations of conjugate gradient using polynomials of order 10. We simulate hardware with one MPI rank per node. Our results are for 9,216 elements and we present the residuals after the 101st iteration.

We plot the residuals using different allreduce algorithms in Figure 14. To better see the small changes between allreduce algorithms, we define zero as the residual of SimGrid's default allreduce algorithm. One algorithm in particular resulted in a residual an order of magnitude smaller every time: smp_rsag. If this algorithm were plotted along with the rest, then the distinction between all other algorithms would vanish. We visualize this difference by tabulating smp_rsag's residual against the default and least accurate residuals in Table 8.

The outlier smp_rsag has smaller error across the various different simulated network topologies we used: 16 and 72-node 2D tori and 16 and 72-node fat trees, as well as different problem sizes. Further, executing Nekbone on a single node with 16 MPI ranks natively using OpenMPI yielded the same result for each of OpenMPI's seven allreduce algorithms. Thus, we predict the cause is the interaction of Nekbone and SimGrid but further investigation is needed.

Even with this outlier, there are just four different results for all SimGrid algorithms. Comparing this to the combinatorial number of acceptable MPI summation orders (4.1) shows

Table 7. Descriptions of allreduce algorithms used to generate Figure 14. "smp" stands for symmetric multiprocessing—inter-node communication is separate from intra-node [190].

Mnemonic	Description
default	naïve reduce then broadcast
impi	use intel mpi selector
mpich	use MPICH selector
mvapich2	use MVAPICH2 selector
mvapich2_rs	rdb for small messages, else reduce-scatter
mvapich2_two_level	smp intra-node, mpich inter-node
ompi	use OpenMPI selector
ompi_ring_segmented	OpenMPI ring algorithm
rab	Rabenseifner
rab2	variation of rab (alltoall then allgather)
rab_rsag	variation of rab (reduce-scatter-allgather)
rdb	recursive doubling
redbcast	reduce then broadcast
smp_binomial	binomial tree
smp_binomial_pipeline	binomial tree with 4096 byte pipeline
smp_rdb	recursive doubling
smp_rsag	reduce-scatter-allgather

Table 8. The smp_rsag algorithm is an outlier. Several algorithms match the default and worst, as indicted in Figure 14.

Allreduce Algo. Rank	Residual
Best (smp_rsag)	$1.616\,306\,278\,792\,575 \times 10^{-8}$
Default	$14.082\,603\,491\,982\,575 \times 10^{-8}$
Worst	$14.082\,603\,491\,982\,647 \times 10^{-8}$

there is need for more nondeterminism when running simulations to get a more complete idea of the space of potential results of HPC codes.

4.6 Related Work

Concern over the accuracy of floating-point summation is as old as floating-point arithmetic itself [202]. Further developments with analytical and statistical bounds for various numerical algorithms is also a well-established area [95] and is broadly contained under the field of numerical analysis. Tools such as FPTaylor [178] and Daisy [55] use sophisticated methods such as symbolic Taylor expansions or SMT solvers, to calculate tight bounds on FP expressions. However, rounding errors as they apply to MPI reduction operations is less widely studied.

Our main influence is work by Chapp et al. on the error of MPI collective operations [38]. This work provides empirical evidence of FP error effects depending on the amount of

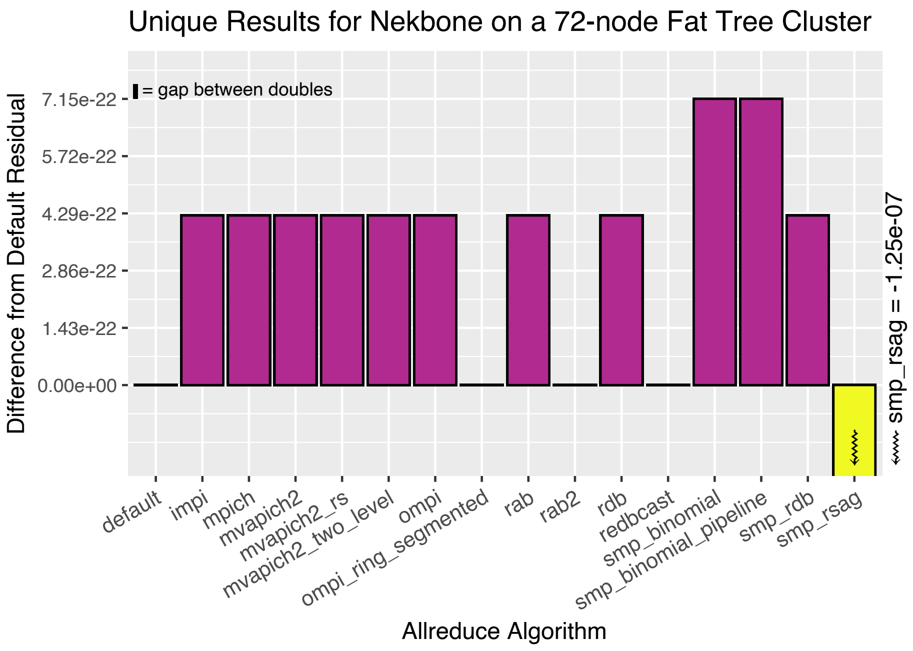


Figure 14. Each bar represents the same residual for the conjugate gradient step of Nekbone across 50 runs. We annotate the distance between two consecutive double-precision floats at the scale of the residual. We cut off the y-axis for otherwise observable differences between other algorithms and smp_rsag would vanish. This shows four unique results among 16 allreduce algorithms.

concurrency as well as reduction-tree shape. Their work advocated a need for dynamic selection between summation algorithms. We refine Chapp's model of MPI reduction error by considering random associations. We find for simulated input—both of our work looks at $U[-1000, 1000]$ floats—random associations results in a lower expected summation error.

Other work regarding MPI [196] looks into MPI collective errors as they apply specifically to the convergence of the conjugate gradient method applied to a power grid analysis.

While our work focuses on analyzing error from unmodified HPC applications, other researchers take this further and strive for bit-level reproducibility. One prominent project in this domain is an algorithm for parallel reproducible summation, [65] and more generally the Reproducible BLAS project [64]. We wish to analyze MPI error compared to ReproBLAS in the future. A different project seeking bit-level reproducibility has been undertaken by Arteaga et al. [7]

Common approaches to reduce summation error include Kahan's compensated summation [107] and prerounding [166]. Both of these can decrease summation error but have performance impacts. Neither is provided by any commonly-used MPI implementation.

SimGrid has also been used to check reproducibility by Hoffeins et al. [98] However this focuses on dynamic loop scheduling whereas we look at only the MPI reduction algorithm itself.

4.7 Conclusion and Future Work

Our work begins a more detailed analysis of summation error with MPI reduction algorithms. We looked the errors for four different families of reduction strategies for a simple `MPI_Reduce` as well as the space of possible solutions for one HPC proxy application using `MPI_Allreduce`, Nekbone. We showed that, at least for some common distributions, the reduction tree shape has a greater effect on the summation error than the order of the array. We also saw that left-associativity typically results in larger error compared to generating a random ordering and reduction tree. This occurs because the distributions we used result in summands of roughly the same magnitude and left-associativity causes partial sums to be much larger than the summands. We demonstrated that analytical error bounds can vastly overestimate summation error. The probability of each partial sum resulting in the worst-case error is not realistic for most problem domains. So, our statistical analysis produces more practically useful expected error.

The problem of analyzing complex HPC applications for error remains unsolved. As mentioned in Section 4.1.1, our experiments with independent, identically distributed random numbers result in relatively predictable behavior. However, distributions of FP numbers in HPC codes are far from random. For example, a singular matrix effectively cannot be generated by randomly generating its elements. A promising future direction would be to generate both more realistic and pathological random FP numbers.

The problem of generating realistic reduction strategies is another interesting direction of research. The most precise description of MPI reduction state space would depend on both network topology and reduction algorithm. However, some refinements to our model could be made: for example, it is unlikely any MPI reduction algorithm would have a large tree height; parallel reduction algorithms strive to complete in $O(\log(n))$ parallel steps. Rémy's procedure generates trees that have an asymptotically different height: on average $4\sqrt{n/\pi}$ [113]. We investigated this somewhat and did not find a correlation between the height of the reduction tree and summation error (the correlation coefficient was between 0.2 and -0.16 for the distributions we sampled). However, it remains to be seen whether generating realistic trees has a significant effect on error.

Beyond this, MPI programs typically have highly-structured reduction schemes. It would be interesting to investigate multiple parallelism schemes and their effect on results. For example, a vector with 10^7 elements may only be reduced across a few dozen nodes, and inter-node summation may be nondeterministic or deterministic depending on the shared memory parallelism scheme. This limits the state space compared to our random generation via Rémy's procedure.

One challenge we note in this work on reproducibility is how hard it can be to get different answers! Small problem sizes, process counts, and simple examples usually result in identical answers. On one hand, this is evidence of the great work by the MPI and SimGrid developers: most MPI implementations are deterministic when run on the same machine with the same input. However, since the standard does not guarantee reproducibility, we wish to explore the entire state space of acceptable answers for a given application by varying the topology and input space to expose potential discrepancies when running at a larger scale or on a different architecture. This could be done by modifying MPI reduction algorithms to inject intentional nondeterminism. Another direction could focus on amplifying errors by using highly numerically unstable input. Both of these strategies are not generalizable to all applications but may give insight into the exact conditions upon which an HPC program can fail.

While reduction operations are important, numerical algorithms consist of more than just adding up numbers. High-performance codes depend on computational building blocks (referred to as kernels) containing many FLOPs but also carefully-tuned numerical approximations. This chapter also considered the oftentimes overlooked behavior of subnormal arithmetic for summation error. In the next chapter we analyze subnormal arithmetic more deeply. We then look at a couple important numerical kernels and carefully analyze both performance and accuracy for a wider variety of numerical routines.

CHAPTER V

SCALABLE ERROR ANALYSIS FOR FLOATING-POINT PROGRAM OPTIMIZATION

5.1 Introduction

As we saw in Chapter IV, the predicted floating-point error of an operation may be much worse than its average case. On the other hand, sometimes the converse is true: for poorly-conditioned systems FP error may be orders of magnitude *worse* than the usual models. This may arise for several reasons, which fall under the broad term *numerical instability*. We briefly investigated numerical instability when calculating which FP numbers will overflow when computing $1 \oslash x$ in Section 2.3.1 (recall that $\oslash$ is FP divide). This chapter investigates more cases where numerical instability can arise, and addresses this for inner products.

To compute the predicted error of a FP program, there exist a handful of different floating-point (FP) error analysis tools such as Gappa [59], Daisy [55], Fluctuat [134], and FPTaylor [178]. These take as input some FP program which may be written an annotated C or Ada program (Fluctuat) or a domain-specific language (Gappa, Daisy, or FPTaylor). Each of these tools use sound overapproximations to bound floating-point error. *Sound* under this context means the true error will never be higher than the predicted error. These tools operate on small FP computational kernels and are all static analyzers: given a sequence of floating-point operations (FLOPS) and their ranges, these tools return error bounds. We give an example input for these tools:

1. A function $f : \mathbb{R}^3 \to \mathbb{R}$. The tool then generates its corresponding FP function $\hat{f}$.
2. Input ranges, e.g., $x, y, z \in [0.001, 1000.0]$.

The output is a bound on the error of $\hat{f}$, e.g., a bound on the absolute error $|f(x, y, z) - \hat{f}(x, y, z)| \leqslant 10^{-14}$.

Kernels in this setting refer to simple subroutines which make up the key components of a computer program. Not only this, but typical scientific codes will spend nearly 100% of their execution time in such kernels. However, these tools are meant for relatively small problem sizes—they do not scale well for programs with more than around 100 FLOPs. This is at odds with HPC kernels, which are designed to work for problems requiring 10^{15} (peta) FLOPs or more. To wit, neither Daisy nor FPTaylor support loops or conditionals in their domain-specific FP languages. Scaling highly-accurate sound error analysis to large problem sizes is a challenging and unsolved problem. Instead, heuristics are used by numerical analysts in the form of "best practices" to improve either performance or numerical stability (ideally both).

These best practices and error analyses may sometimes be overly pessimistic and sometimes may be overly optimistic. We begin this chapter by demonstrating how FLOPs can be poorly behaved for subnormal arithmetic, then provide refined error bounds to account for this. Next, we analyze some typical heuristics and optimizations used by numerical analysts and quantify them, thus providing concrete error bounds behind vague statements such as "numerical stability." We also collect the error and performance characteristics for common high-performance machine instructions to better understand this performance-accuracy tradeoff. We use vector normalization as a motivating example because of its ubiquity in numerical computation and computer graphics.

5.2 Two Kinds of Floating-Point: Normal and Subnormal

The error resulting from a FLOP is an incredibly complex function of its inputs. To make this tractable, typically FP error is modeled as follows [81, 96]. Let $\odot$ be a floating-point operation and $\cdot$ its corresponding operation over $\mathbb{R}$. Further, let flt be some FP format with p bits of precision. For example, for IEEE 754 single-precision, $p = 24$ and for double-precision $p = 53$. Unless otherwise specified, single and double-precision refers to the IEEE 754 binary32 and binary64 formats, respectively. Then we define *machine-epsilon* as 2^{-p-1}. Assuming a rounding-to-nearest rounding mode, the relative error of an operation is bounded by $u = \varepsilon/2$. We call this u the *unit roundoff*[1]. As we saw in Section 2.8.2, a common model of the relative error of a FLOP is modeled by

$$x \odot y = (x \cdot y)(1 + e) \tag{5.1}$$

such that $|e| \leqslant u$. The reader may confirm that $|e|$ is the same relative error as given in Definition 2.4.

However, for numbers very close to zero this is not an adequate model. Here, as before, $\otimes$ represents floating-point multiplication. For example, let $x = 10^{-4}$ and $y = 3 \times 10^{-40}$. For single-precision, y is a *subnormal* number. We expect a number near $x \times y = 3 \times 10^{-44}$. However,

$$x \otimes y = 2.942727 \times 10^{-44} = (x \times y)(1 - 1.9091074972 \times 10^{-2}), \tag{5.2}$$

[1]Some authors define ε as the unit roundoff as well.

the relative error of this multiplication is six orders of magnitude larger than $u = 2^{-24} \approx 5.96 \times 10^{-8}$. This is because subnormal floats have less precision. It is known that numbers close to 0 have high potential error; beyond this, for many numerical algorithms such systems also indicate an ill-conditioned system. Put bluntly: FP error analysis using (5.1) can be *unsound*.

Soundness is important because underapproximating error is potentially much more dangerous than overapproximating error. While this is a broad statement, consider the problem of ensuring two aircraft do not crash into each other mid-air. If the distance between them is overapproximated, the aircraft needlessly steer clear of each other. If it is underapproximated, then a collision may occur even if one was predicted to not happen. This is not a simple theoretical curiosity; researchers at NASA care deeply about this problem [142, 191].

With this in mind, we use a more accurate model of FP error analysis as described in [84, Eq. (11)]. This is done by specifying an *absolute error* in addition to the relative error. Specifically, given an FP format flt, alongside u we specify an additional term $\delta = \varepsilon \times 2^{e_{\min}}$, where $e_{\min}$ is the smallest exponent in flt. Using our example of single precision, $e_{\min} = 2^{-126}$ and so $\delta = 2^{-149}$. Put together, the refined model is

$$x \odot y = (x \cdot y)(1 + e) + d \tag{5.3}$$

such that $|e| \leqslant u$ and $|d| \leqslant \delta$.

In most cases it is easier to model just one of the relative (e) or absolute (d) error terms. In our current example, assume $e = 0$ so

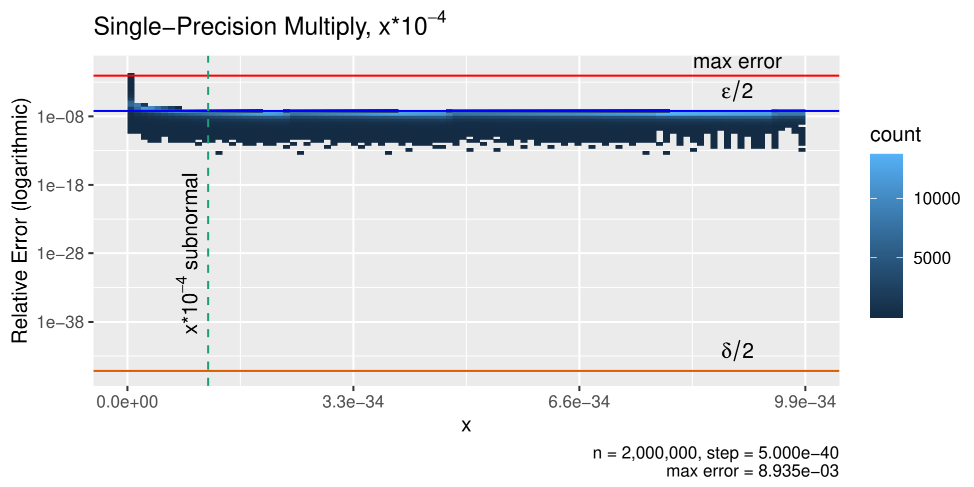
$$d = x \otimes y - x \times y = 2.942727 \times 10^{-44} - 3 \times 10^{-44} = -5.727322 \times 10^{-46}$$

and so

$$|-5.727322 \times 10^{-46}| \leqslant 2^{-149} \approx 1.401 \times 10^{-45}.$$

Figure 15 illustrates further these examples for single precision floats. Both plots show a range of normal and subnormal values: to the left of the dotted lines show subnormal values and to the right show normal values. Further, we do not sample *every* single-precision float but rather every 5×10^{-40} to show the behavior for a larger range.

Figure 16 shows a similar picture, only for the square of a value; this will be useful to keep in mind as we move on to the analysis of vector normalization. Another difference is that Figure 16 is dense in its inputs—every FP number in its range is plotted. Both figures show the



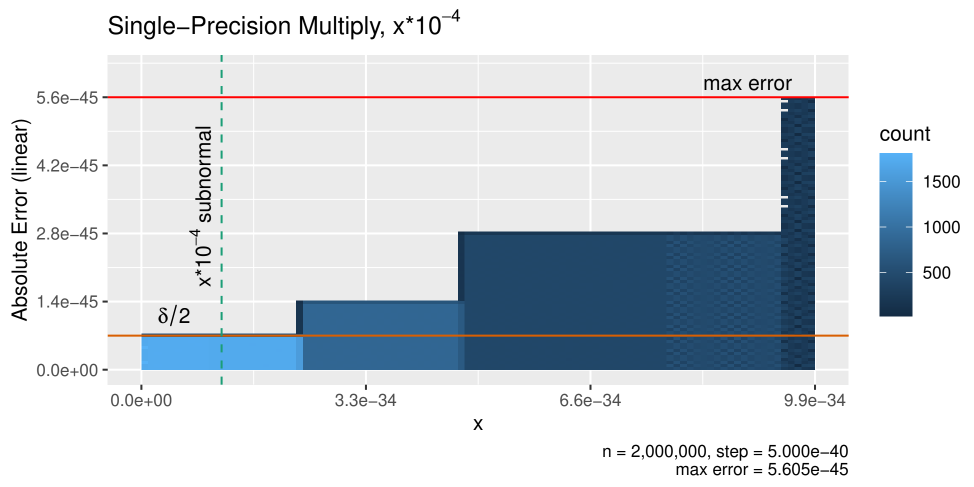


Figure 15. Comparison of absolute and relative errors around the cusp between normal and subnormal floats. The range chosen is one which contains products (x^2) which straddle the cusp between subnormal (to left of the dotted lines) and normal floats (to the right).

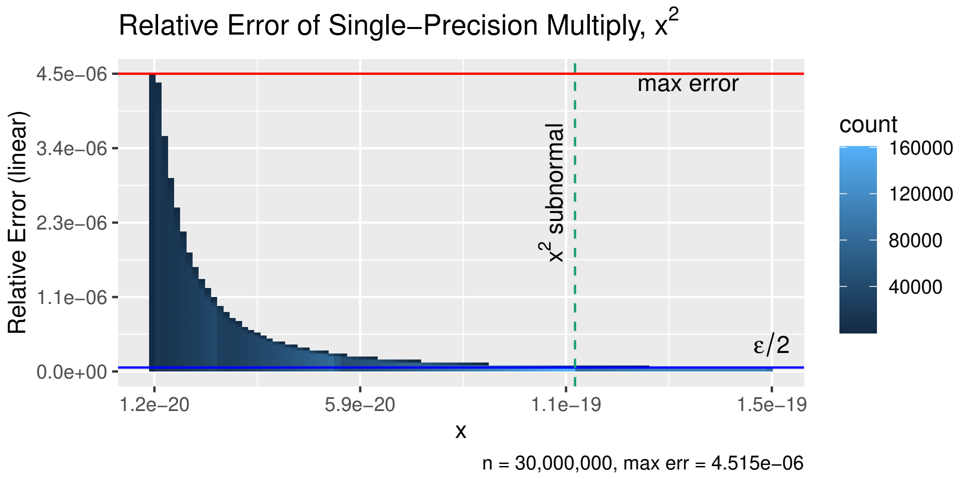


Figure 16. Relative error for $x \otimes x$ where x is a floating-point number close to 0. The ranges plotted are exhaustive. That is, this plot shows the errors for every single single-precision float in the range $[1.2454331 \times 10^{-20}, 1.5332933 \times 10^{-19}]$.

fill color as density of error, but this gives an incomplete picture. As x^2 becomes subnormal, the relative error increases far beyond $\varepsilon/2$.

To get a better idea of the distinction between normal and subnormal, we plot both distributions of error in Figures 17 and 18. Measuring the error in units in the last place (ulps), a FLOP using the round-to-nearest mode for normal numbers has relative error in the range $[-\varepsilon/2, \varepsilon/2]$ ($[-0.5, 0.5]$ ulps). The subset of values for which x^2 are normal is plotted in Figure 17 and for x^2 subnormal are plotted in 18.

These figures show that careful consideration must be done for operations where subnormal numbers are likely to appear. This and our desire for soundness lead us to derive sound error bounds for common computational kernels. We begin with inner products because they form the basis of most other key numerical methods.

5.3 Refining Error Analysis of Inner Products

Nicholas Higham is one of the premier researchers in numerical analysis. His book [96] develops error analysis for a wide variety of numerical methods and the results build nicely off of simple numerical building blocks such as inner products. Using (5.3), we update Higham's error analysis for inner products to include subnormals [96, Lemma 3.1].

Inner products are an important notion in vector spaces to bestow a vector space with a notion of "length" or "angle" between vectors. However, for our purposes, we may think of *inner*

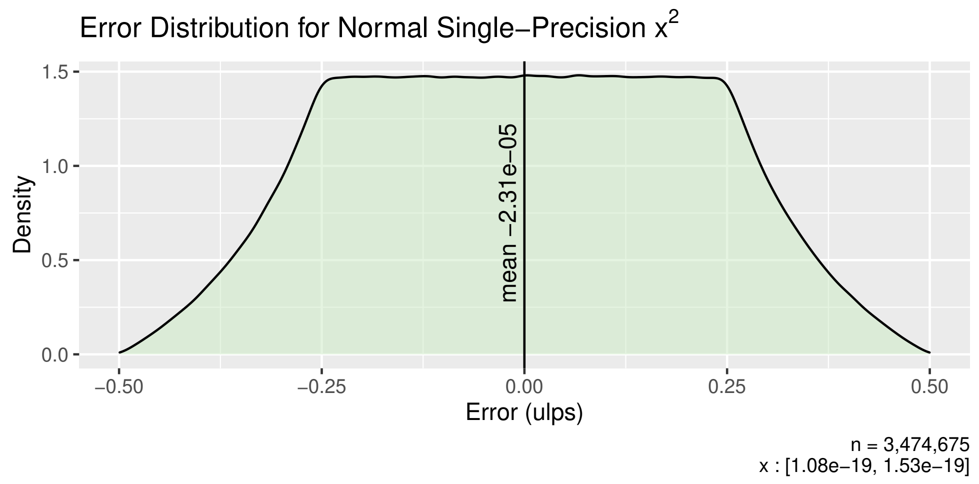


Figure 17. Error in ulps for small, but normal, floating-point error square, i.e., $x \otimes x$.

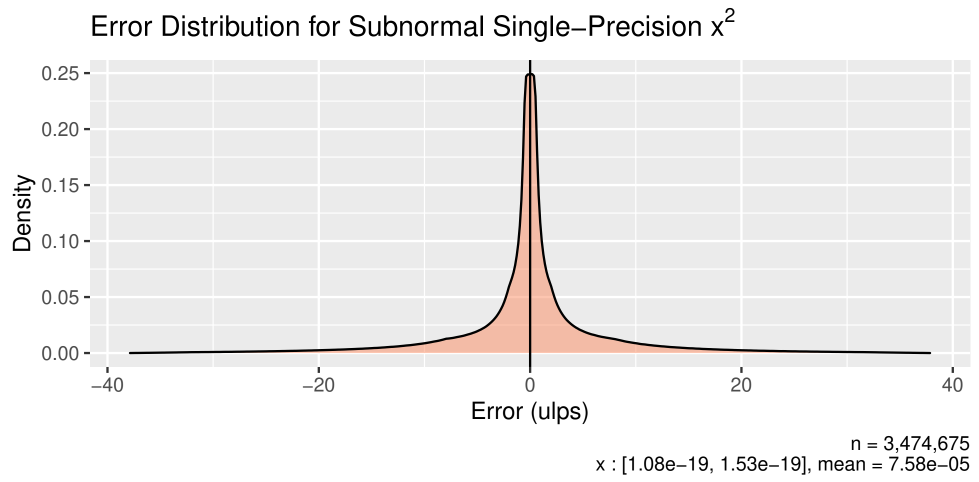


Figure 18. Error in ulps for small, subnormal, floating-point square, i.e., $x \otimes x$.

product and *dot product* interchangeably. This becomes clearer symbolically. Suppose x and $y \in \mathbb{R}^n$ are real-valued vectors of length n. Then the inner product is

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i.$$

We begin deriving error analysis using (5.3) for each partial sum by defining the approximations at each step assuming left-associative summation (though the bound is the same regardless of summation order). So,

$$\begin{aligned}
\hat{z}_1 &= x_1 y_1 (1 + \eta_1) + d_1 \\
\hat{z}_{k+1} &= \left(\hat{z}_k + (x_{k+1} y_{k+1})(1 + \eta_{k+1}) + d_{k+1}\right)\left(1 + \eta'_{k+1}\right).
\end{aligned}$$

The final inner product is then $\hat{z}_n$. We do not need d_k terms for the additions because $d = 0$ for floating-point addition. This is because the error from floating-point point addition is itself a floating-point number [29, Lemma 5.2].

To begin to simplify this, note that each η_k and d_k need not be distinguished; the largest errors occur for $\eta = \pm u$ and $d = \pm\delta/2$. And similarly, since the largest errors are for $-\delta/2$ and $-u$ or $\delta/2$ and u, we combine the signs for η and d into just $\pm\eta$. Expanding out, we get

$$\begin{aligned}
\hat{z}_2 &= x_1 y_1 (1 \pm \eta)^2 + x_2 y_2 (1 \pm \eta)^2 + d(1 \pm \eta) + d(1 \pm \eta) \\
\hat{z}_3 &= x_1 y_1 (1 \pm \eta)^3 + x_2 y_2 (1 \pm \eta)^3 + x_3 y_3 (1 \pm \eta)^2 \\
&\quad + d(1 \pm \eta)^2 + d(1 \pm \eta)^2 + d(1 \pm \eta) \\
&\vdots \\
\hat{z}_n &= x_1 y_1 (1 \pm \eta)^n + \sum_{i=2}^{n} x_i y_i (1 \pm \eta)^{n-i+2} \\
&\quad + d(1 \pm \eta)^{n-1} + \sum_{i=2}^{n-1} d(1 \pm \eta)^i.
\end{aligned}$$

To simplify this, we use [96, Lemma 3.1] which allows us to substitute the $(1 \pm \eta)^i$ terms since the error is greatest when $\eta = \pm u$. That is, provided $|\eta| \leqslant u$, $\rho_i = \pm 1$, and $nu < 1$, then

$$\prod_{i=1}^{n} (1 + \eta_i)^{\rho_i} = 1 + \theta_n$$

where

$$|\theta_n| \leqslant \frac{nu}{1 - nu} = \gamma_n. \tag{5.4}$$

So our z_n becomes

$$\hat{z}_n = x_1 y_1 (1+\theta_n) + \sum_{i=2}^{n} x_i y_i (1+\theta_{n-i+2}) \tag{5.5}$$
$$+ d(1+\theta_{n-1}) + \sum_{i=2}^{n-1} d(1+\theta_i).$$

To simplify and to permit an arbitrary summation order, we overapproximate the lower-order θ terms. Also, we replace each d with its maximum value of $\delta/2$ and so rewrite (5.5) as

$$\hat{z}_n \leqslant \sum_{i=1}^{n} x_i y_i (1+\theta_n) + \sum_{i=1}^{n} \frac{\delta}{2}(1+\theta_{n-1})$$
$$= \sum_{i=1}^{n} x_i y_i (1+\theta_n) + n\frac{\delta}{2}(1+\theta_{n-1}). \tag{5.6}$$

Therefore, an error bound for inner products which works for subnormal and normal FP values is

$$|\langle x, y\rangle - \mathrm{flt}(\langle x, y\rangle)| \leqslant \gamma_n |x| \cdot |y| + n\frac{\delta}{2}(1+\gamma_{n-1}), \tag{5.7}$$

which is a sound version of Higham's Equation (3.5) [96].

This allows us to make *sound* overapproximations of the error of inner product. In most cases, the δ terms are insignificant. However, using (5.3) as our model of floating-point arithmetic ensures that as long as floating-point exceptions are now thrown, floating-point error by our estimation will never be less than the true error.

Beyond inner products, numerical programs contain many different highly-optimized kernels. It is through the careful and creative work of previous researchers that we, in general, trust these computations. Some examples can be seen not only in libraries like the Basic Linear Algebra Subroutines (BLAS) [23] but other math libraries [84, 58, 21]. In many cases there are multiple versions of an operation, those optimized for speed or accuracy such as `clartg` versus `crotg` in BLAS [21, p. 229]. In the next section we describe how numerical programs are constructed from simpler ones, including inner products. Now that we have a sound bound for inner products, wee look at how other libraries work with inner products and the next natural extension: the norm (or length) of a vector.

5.4 The Building Blocks of Numerical Programs

Numerical programs, as with other computer programs, are formed from composition of smaller programs. One way this is done is through what are called *computational kernels* or small subprograms which perform some arithmetic operation. We focus on one widely used collection called BLAS. These routines are divided into three levels, called, unsurprisingly 1, 2, and 3. These correspond to the computational complexity of the algorithm. That is, given inputs of size n, BLAS-1 routines are $O(n)$, BLAS-2 routines are $O(n^2)$ and BLAS-3 are $O(n^3)$.

For example, computing the length of some vector x can be done either in a straightforward loop (further explained in Section 5.5) or via the BLAS routine `dnrm2`. The advantages to building a program from BLAS instead of from scratch are twofold:

1. BLAS implementations such as OpenBLAS [198] have aggressively-optimized routines;
2. BLAS implementations are very careful to ensure numerical stability for the greatest range of FP numbers.

Conversely, one may not want to use BLAS. Nothwithstanding issues of software engineering from adding additional dependencies to a project, sometimes the accuracy is more than is needed or the overhead of calling BLAS is too high. This may be the case with computing a vector norm for small vectors in visualization or computer graphics: BLAS routines are meant to be called relatively small numbers of times on large inputs, though a standard called Batched BLAS seek to alleviate this limitation [67].

We motivate the importance of composition of numberical routines by considering the QR decomposition of a matrix [54]. QR decomposition is widely used in numerical analysis and statistics because it provides an easy way to compute the eigenvalues and eigenvectors of a matrix. It has a wide application domain, and even more recent work finds has applications Principal Component Analysis (PCA) [174]. QR factorization of a matrix $A \in \mathbb{R}^{n \times n}$ requires $O(n^3)$ operations. In the BLAS hierarchy, this would be considered a Level 3 subroutine, though the BLAS standard does not specify such a complex operation. Instead, depending on which of the three families of algorithms for QR decomposition (Householder reflectors, Givens rotations, or modified Gram-Schmidt), it may look completely different. In practice, libraries such as LAPACK [3] implement QR and other algorithms by calling these lower-level BLAS routines. To keep the problem tractable, we focus on simpler operations, specifically at the BLAS-1 level. This gives the dual benefit of being possible to analyze completely as well as being widely applicable. The example we choose is normalizing a vector for two reasons: it is a critical step in many numerical programs and the two operations, division and square-root, add complexity to error analysis. This provides a more realistic application of FP error analysis.

Listing 5.1 A simple implementation of vector normalization.

```
// Compute norm2 as ⟨x,x⟩
norm2 <— 0.0
for each x_i in x
do
    norm2 <— norm2 + x_i * x_i
end
rnorm <— 1.0/sqrt(norm2)
// Scale each element of x
for each x_i in x
do
    x_i <— x_i * rnorm
end
```

5.5 Normalizing a Vector

While there are many norms of vectors, normalization refers almost exclusively to the *Euclidean norm* or the *2-norm*, denoted $\|x\|_2$. Symbolically, if $x \in \mathbb{R}^n$, then

$$\langle x, x\rangle = \|x\|_2^2 = \sum_{i=1}^{n} x_i^2.$$

Normalizing a vector x creates a new vector q such that the length of q is 1. Typically, this normalization is done in place to avoid memory allocation. We provide a simple algorithm for vector normalization in Listing 5.1.

To see the complexity hidden in such a simple operation, consider a case when $x_i \otimes x_i$ overflows but the overall length of the vector is finite. For example, consider the vector $x = [10^{175}, 10^{175}]^\top$. Then $10^{175} \otimes 10^{175} = \infty$. However, the norm of the vector is $\sqrt{2} \times 10^{175}$ which is finite in double-precision FP. BLAS solves this by storing a scale factor during the accumulation, as shown in Listing 5.2. The drawback to using `dnrm2` is its speed: the conditionals inside the loop results in slower code.

Already, for a simple operation we are confronted with an accuracy and speed tradeoff. Using BLAS, we have two different possible ways to do a vector normalization. The actual API of BLAS is more complex; we simplify for clarity in Listing 5.3. Note that the BLAS implementation of `ddot` does not check for overflow; moreover if x is the vector of all zeros then Version 2 will result in division by 0. This is just one of many considerations numerical analysts must keep in mind as they write high-performance scientific codes. One potential solution is using static analysis from automated tools to detect whether overflow can occur. However, these techniques

Listing 5.2 CBLAS version of the `dnrm2` vector norm from v2.6 of GNU Scientific Library [79]

```c
double scale = 0.0;
double ssq = 1.0;
INDEX i; /* BASE and INDEX are integer types */
INDEX ix = 0;
if (N <= 0 || incX <= 0) {
    return 0;
} else if (N == 1) {
    return fabs(X[0]);
}
for (i = 0; i < N; i++) {
    const BASE x = X[ix];
    if (x != 0.0) {
        const BASE ax = fabs(x);
        if (scale < ax) {
            ssq = 1.0 + ssq * (scale/ax) * (scale/ax);
            scale = ax;
        } else {
            ssq += (ax/scale) * (ax/scale);
        }
    }
    ix += incX;
}
return scale * sqrt(ssq);
```

Listing 5.3 Vector normalization using BLAS.

```
// Version 1: Safe from overflow but slower
norm <- dnrm2(x)
rnorm <- 1.0 / norm
dscal(x, rnorm)
// Version 2: May overflow, but faster
norm2 <- ddot(x)
rnorm <- 1.0 / sqrt(norm2)
dscal(x, rnorm)
```

are limited in the problem size they are able to handle. We begin by describing the state of the art in static analysis of numerical codes and work to improve this.

5.6 On the Scalability of Static Analysis

We mentioned several static FP error analysis tools at the beginning of this chapter: FPTaylor, Fluctuat, Daisy, and Gappa. More recently, a tool called Satire [56] works to analyze more FLOPs. Examples in the original paper showed scalability up to 300,000 FLOPs. We use our running example of normalizing a vector using Satire and FPTaylor to investigate potential scalability. Each of these projects have the advantage of working on many different FP programs. While Gappa and Fluctuat work on general purpose numerical codes, they require more user input to guide proofs and so are not automated.

Beyond this, FPTaylor, Daisy, and Satire operate only on so-called "straight-line" code: no loops or conditionals. While in theory, one may generate input for arbitrary vector lengths, in practice this makes using these tools for large inputs inconvenient. We investigated FPTaylor and Satire to determine how well they scale to large numbers of inputs in the case of vector normalization. From our results in Section 5.3, we use (5.7) to generate error bounds as well. The benefit to this is it is computationally cheap: the cost to analyze the error is independent of the size of the input vector.

We investigate both the scalability of FPTaylor and Stire and compare it with a combined approach, using (5.7) to compute the error bound for $\langle x, x \rangle$, then using the result as input to FPTaylor, allowing it to compute the error of only 2 FLOPs—FP divide ($\oslash$) and square root ($\sqrt[\text{fp}]{\cdot}$)—instead of $2n + 1$. We arrange the results in Table 9.

As we see, with larger n existing tools begin to break down. While we used a fixed range of $[0.001, 1000]$ in all cases, for larger n this is considered a large range: error analysis takes longer because of the potentially many subintervals over which the error must be bounded. We found the best results with a combination of the error bounds given in (5.7) with some caveats.

In any case, these are far greater error bounds than occur in the average case. This is why probabilistic error analysis is an exciting research area. One thing to keep in mind is, as shown in Figures 17 and 18, the relative error increases for subnormals. For example, FPTaylor encounters errors if the input domain is set to single-precision and the ranges are set to $[5 \times 10^{-20}, 1 \times 10^{-14}]$: further evidence that subnormal arithmetic causes problems on even small numerical codes.

Table 9. Performance and error results from various tools and analyses to compute $1 \oslash \sqrt[\text{fp}]{\langle x, x\rangle}$. In all cases each x_i is a double-precision float with $x_i \in [0.001, 1000]$.

Approach	Size	Runtime (s)	Max. Absolute Error
Satire	50	44	5.5505×10^{-8}
Satire	100	178	2.8618×10^{-6}
Satire	500	13 593	1.0270×10^{-2}
FPTaylor	50	195	2.7434×10^{-4}
FPTaylor	100	433	7.0310×10^{-4}
FPTaylor	500	5505	∞
(5.7) & FPTaylor	50	< 0.1	7.9172×10^{-1}
(5.7) & FPTaylor	100	< 0.1	1.1295
(5.7) & FPTaylor	500	< 0.1	2.4825×10^{-6} *
(5.7) & FPTaylor	10^6	0.6	4.4631×10^{-3} *
(5.7) & FPTaylor	10^9	9.2	4.4631×10^{-3} *

* Required taking the maximum of two analyses: one for $[0.001, 1]$ and one for $[1, 1000]$.

We saw in Chapter IV that predicted error bounds may be orders of magnitudes greater than the actual program behavior. Recent work in the field of numerical analysis, notably Higham [97] and Ipsen [103] use probabilistic error analysis. While there has been criticism of probabilistic numerical error analysis [109], these works are useful as long as their underlying assumptions are met. Considering this, we next move to analyzing the reciprocal norm of a vector from a different angle: analyzing the performance-accuracy tradeoff between using fast approximations and slower, more correct solutions.

5.7 Towards a Cost Semantics

The reciprocal square root is sufficiently commonly used that many architectures provide either an implementation or approximation in hardware. We begin to develop a cost semantics to compare the performance-accuracy tradeoff for reciprocal square root (also called `rsqrt`) as well as other vector operations. We present some runtime costs in Table 10 on modern CPU architectures. We describe here a few different versions of $\texttt{rsqrt}(x) = 1/\sqrt{x}$ which is defined for most $x > 0$. We say *most* $x > 0$ because there are some subnormal floating-point $x > 0$ for which $1.0 \oslash x = \infty$ (see Listing 2.3).

The Intel figures are published in their external documentation [102] and the ARM performance values were taken from Fujitsu's reference manual. There are many challenges when creating cost semantics. One challenge is the documentation themselves: many combined operations such as `_mm512_reduce_add_pd`, which performs a reduction on 8 double-precision

Table 10. Cost semantics for reciprocal square root, square root, add, multiply, and divide. Throughput is in cycles per instruction (CPI) and error is the upper bound on relative error (RE). Data are gathered from the Intel [102] and Fujitsu [78] reference manuals.

Architecture	Prec.	Mnemonic	Op.	Latency	Thruput[1]	max(RE)
Intel Skylake	double	`mm512_add_pd`	$\oplus$	4	0.5	$\varepsilon/2$
Intel Skylake	double	`mm512_div_pd`	$\oslash$	23	16	$\varepsilon/2$
Intel Skylake	double	`mm512_mul_pd`	$\otimes$	4	0.5	$\varepsilon/2$
Intel Skylake	double	`mm512_rsqrt14_pd`	$1 \oslash \sqrt[\text{fp}]{\cdot}$	9	2	2^{-14}
Intel Skylake	single	`mm512_rsqrt14_ps`	$1 \oslash \sqrt[\text{fp}]{\cdot}$	9	2	2^{-14}
Intel Skylake	single	`mm256_rsqrt_ps`	$1 \oslash \sqrt[\text{fp}]{\cdot}$	4	1	$\frac{3}{2} \times 2^{-12}$
Intel Skylake	single	`mm512_sqrt_ps`	$\sqrt[\text{fp}]{\cdot}$	19	12	$\varepsilon/2$
Intel Skylake	double	`mm512_sqrt_pd`	$\sqrt[\text{fp}]{\cdot}$	31	24	$\varepsilon/2$
Intel Skylake	double	`mm512_invsqrt_pd`	$1 \oslash \sqrt[\text{fp}]{\cdot}$	ND[2]	ND	ND
Arm A64FX	double	(512 bit) `fadd`	$\oslash$	4	ND[3]	$\varepsilon/2$
Arm A64FX	double	(512 bit) `fdiv`	$\oslash$	154	ND	$\varepsilon/2$
Arm A64FX	double	(512 bit) `fmul`	$\otimes$	9	ND	$\varepsilon/2$
Arm A64FX	double	(512 bit) `fsqrt`	$\sqrt[\text{fp}]{\cdot}$	154	ND	$\varepsilon/2$
Arm A64FX	double	(64 bit) `frsqrte`	$1 \oslash \sqrt[\text{fp}]{\cdot}$	4	ND	ND
Arm A64FX	double	(64 bit) `frsqrts`	$1 \oslash \sqrt[\text{fp}]{\cdot}$[4]	9	ND	ND

[1] Throughput is measured in Cycles Per Instruction (CPI). That is, lower is faster.
[2] ND means Not (officially) Documented
[3] Fujitsu does not provide throughput but only pipeline stages.
[4] Performs Newton-Ralphson iteration of $y \otimes 2 \otimes (3 \ominus x \otimes y \otimes y)$ for y the current guess and x the input.

FP numbers, does not have a cost associated with it in official documentation. Some cost semantics may be poorly documented. For example, subnormal arithmetic is usually only said to be "slower," not how much slower. But a larger issue is one of mapping these documented performance results to those on actual hardware: in reality, performance depends on clock speed, workflow, data dependencies, and other proprietary microarchitecture features. This is a research direction we mention in our conclusion. We begin a cost semantics for two popular processor architectures in the HPC space: Intel Skylake and Fujitsu A64FX. Both support vector operations up to 512 bits (8 double-precision floats). We show some operations and their associated costs in Table 10.

One valid complaint of using architecture-specific optimizations is its lack of portability, not only across architectures but between newer versions of the same architecture. Additionally, since the architectures are proprietary it is impossible to know what is happening at the silicon-level. Because of these challenges, the following section analyzes in detail a few different ways to compute an optimized reciprocal square root without relying on hardware-specific features.

5.8 Optimizing Reciprocal Square Root

Reciprocal square root, or `rsqrt`, is an important operation with applications in computer graphics as well as numerical analysis. However, we must not misrepresent the importance of a single operation among a larger workflow. For example, suppose an algorithm operating on a vector of size $n = 10^9$ normalizes a vector. In this case, the dot product and normalization procedure is $O(n)$ FLOPs but the vector normalization is only 2: a division and square root. Even though those two FLOPs take much longer, the runtime is negligible. Therefore, we motivate these use cases mainly for data visualization or matrix operations on small matrices. With this in mind, we consider a few different implementations of `rsqrt` and their error characteristics.

Modern implementations of non-elementary operations (that is, anything beyond $+$, $-$, $\times$, and $/$) are based off many clever strategies. One such strategy is to use Newton-Ralphson iteration, which is an iterative method to find the zeros of a function; i.e., the x such that $f(x) = 0$. Then, given a sufficiently close initial guess x_0, the sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

converges to x. Here f' is the derivative of f. In the case of $f = 1/\sqrt{x}$, the iteration may be done very efficiently without any division operations. We do not derive the iteration here but it is based on the observation that if $y = 1/\sqrt{x}$, then computing the reciprocal square root is equivalent to finding the solution to $f(y) = 1/y^2 - x = 0$. Applying some algebraic manipulations, we get the following iteration to compute `rsqrt`:

$$y_{n+1} = 2y_n(3 - xy_n^2). \tag{5.8}$$

This just leaves the issue of choosing a good initial guess. The convergence speed of Newton-Ralphson iteration depends on such a guess, and functions with more than one solution may not converge to the correct solution if the guess is not sufficiently accurate. These initial guesses are computed many different ways such as precomputed lookup tables or building some Taylor series approximation of the true function. We show the convergence properties for three different initial guesses: using Intel's estimator (Figure 19), a first-order Taylor polynomial (Figure 20) and another quasi-arbitrary guess (Figure 21).

In all of these, the x-axis is the true result of `rsqrt` and the y-axis is the relative error. And so the inputs in each case are in the range $[10^{-1}, 10^3]$. We sample 1,000 points in log-space

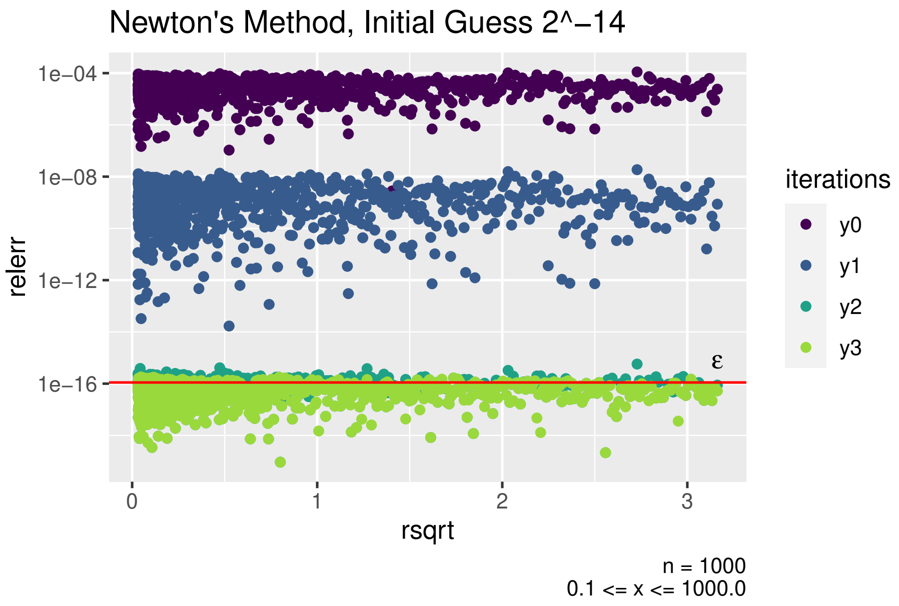


Figure 19. Convergence of $1 \oslash \sqrt[\mathrm{fp}]{\cdot}$ with a good initial guess (relative error at most 2^{-14}). This value was chosen because such an initial guess exists in Intel hardware (see `_mm512_rsqrt14_ps` in Table 10).

(that is, the exponents are uniformly distributed in $[-1, 3]$). We choose this range to see the distinction more clearly; for a larger range the approximations in Figures 20 and 21 require many more iterations. This sort of consideration is why scaling input or lookup tables are used in many numerical routines. The brighter colors indicate more Newton-Ralphson iteration. Specifically, each iteration adds six FLOPs: five multiplications and one addition, some of which can be pipelined. Consulting Table 10, we can begin to analyze the tradeoff between performance and accuracy.

We mention a few things about each figure. Figure 19 is by far the best estimation. We show only three approximations since the error does not improve beyond this (in fact, the solution has pretty much converged by the second iteration). Between Figures 20 and 21, each has better convergence around different values: the initial guesses are best near $1/\sqrt{x} = 0.5/x$ $(x = 0.25)$ $1/\sqrt{x} = 1 - (x - 1)/2$ $(x = 1)$. One could easily choose better approximations if distribution of the input space was known, thus providing further evidence that knowledge of input domain is useful for both program correctness and performance.

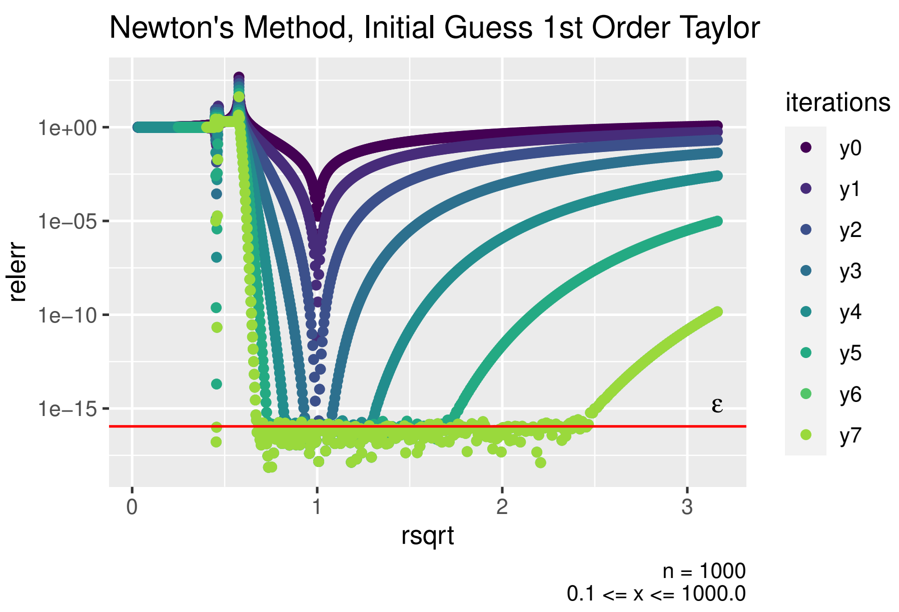


Figure 20. Convergence of $1 \oslash \sqrt[\text{fp}]{\cdot}$ using a first-order Taylor polynomial $T_1(x) = 1 - (x - 1)/2$ to compute the initial guess x_0.

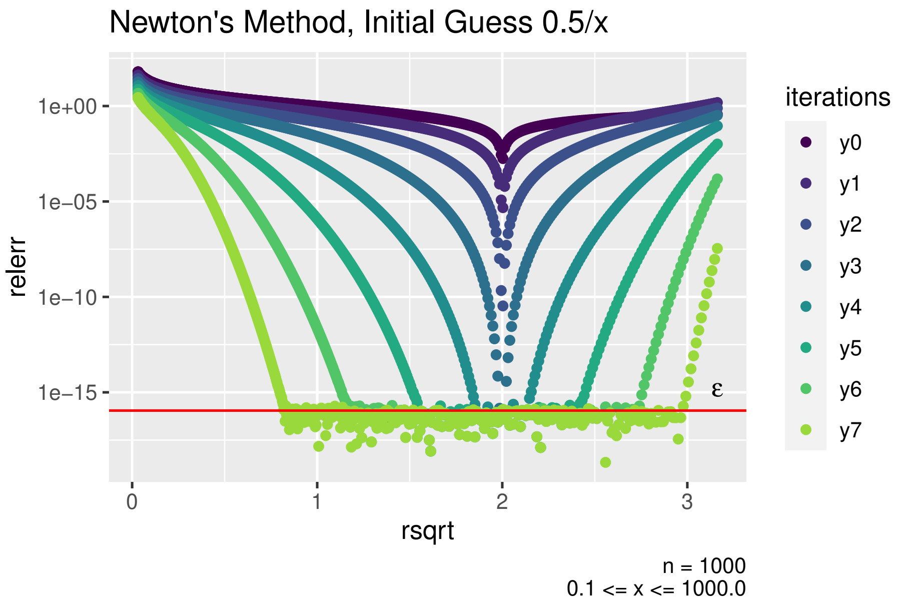


Figure 21. Convergence of $1 \oslash \sqrt[\text{fp}]{\cdot}$ using a different initial guess of $x_0 = 0.5 \oslash x$.

5.9 Related Work

The reciprocal square root is somewhat widely studied: recent work also analyzed `rsqrt` as an application to clustering algorithms [135]. The accuracy-performance tradeoff has been widely studied and the references are too many to list: some examples are those at Lawrence Livermore National Laboratory [117] and at Université de Perpignan [120].

Beyond the automated tools we mentioned throughout this paper like FPTaylor and Satire, Goubault and Putot analyze in detail smaller programs more completely using static analysis [85]. Instead of static analysis, there is also the idea of tracking error during runtime. Nestor Demeure's Ph.D. thesis [63] is based on keeping track of error alongside each FLOP via a higher-precision value. Demeure's tool, Shaman, inserts hooks into floating-point operations and so has some fixed multiple of overhead: errors from single-precision floats are tracked with double-precision, and double-precision with larger-precision MPFR [76] floats. Our analysis is not automatic but instead carefully and completely analyzes algorithm building blocks as opposed to arbitrary programs.

Approximate computing often takes a similar approach by specifying a desired precision with each computation. Approximate computing mostly finds applications in places with severe power constraints such as embedded systems and is usually considered more at the hardware level. A survey of techniques is given by Mittal [138].

There are careful floating-point error bounds written up for LAPACK, for example [22]. The LAPACK authors claim that underflow error is usually worse than round-off error, and when it is not true there this indicates a system in which the underlying problem is not expressed well. Even so, the notion of *ill-conditioned*, typically expressed using a system's *condition number* is not a sufficiently detailed explanation of error. For example, what if flight control software encounters a rare scenario or a sensor mishap produces such an ill-conditioned system? We wish to know precisely under what conditions a system will fail. LAPACK describes error bounds via code fragments with detailed information about individual routines like those previously mentioned [21], but analysis of larger programs remains unsolved.

5.10 Future Work

The question of extending analysis to large programs and large input is still unsolved. To begin, the ranges used in this analysis do not use the terms of (5.3) which handle subnormal arithmetic: those are dozens of orders of magnitude smaller than the minimal value in our experimental analysis ($10^{-3} \otimes 10^{-3}$). This is because of the challenges of reasoning with very small numbers in existing tools. For example, FPTaylor will indicate a potential error if it cannot guarantee $1 \oslash \sqrt[\mathrm{fp}]{x}$ does not result in division by zero. We addressed this by subdividing the intervals manually and dispatching each case to FPTaylor (a comparable tool could also be used). Future work could extend FPTaylor to act as one part of a more complete analysis: this would allow error bounds derived various ways (for example, through manual analysis or other tools) which would be a starting-point for FPTaylor. This would permit a more automated workflow instead of requiring a human to analyze error bounds and split intervals manually.

The next step towards analysis full numerical routines would be composition of larger subroutines. Our analysis focuses on some Level 1 BLAS, but numerical algorithms consist of the composition of Levels 1, 2, and 3 subroutines. The propagation of error across these would truly allow sound scalable analysis of realistic FP programs. This can be done in a similar fashion to Section 5.3 along with mechanization of existing analysis—which currently exists only in mathematical notation—for other BLAS routines. To round out this analysis of BLAS routines, we must support complex arithmetic which is usually ignored in FP error analysis. Fortran has a complex data type built-in, but analysis for C/C++ remains limited. There is a large amount of mathematical theory for complex numbers; our FP error analysis should be generalizable to complex numbers without major difficulty.

5.11 Conclusion

In this chapter, we described a technique for sound, scalable FP error analysis. Scalability was achieved with a combination of sound bounds (via (5.7)) on FP error for simpler operations like inner products, followed by finer-grained analysis of error using FPTaylor or analysis of convergence for Newton-Ralphson iteration for more complex operations like square root and division. We combined these to bound the error on vector normalization. The sorts of detailed, fine-grained analyses outlined in this chapter are only worthwhile for codes which we know will run many times because they require a combination of several techniques to understand the properties of the underlying real number behavior. This answers part two of this dissertation's main question by applying the high-level notion of soundness and sound error approximations to an FP kernel operating on vectors.

While our work did achieve scalability to essentially arbitrary-sized vectors, this comes with a few caveats. For one, the absolute error may not be as informative as relative error—the places which maximal absolute error occur in an interval are likely not the same as where the maximum relative error occurs.

For Section 5.7, we do not have experimental evidence to back up the tabulated cost semantics. But even if this were included, it would be ephemeral. The problem of performance reproducibility is well-known in HPC [148] but a more fundamental problem is that of longevity of research. For example, consider the Intel Xeon Phi coprocessor. This was an architecture focused on high-throughput vectorized instructions and many were installed in supercomputers in the 2010s. We considered including performance characteristics for this architecture since it includes more `rsqrt` vector operations, but they are no longer being manufactured and in 10 years there will be no more Phi coprocessors in use. This renders the myriad performance results of the Phi coprocessors less useful to the community.

Numerical codes require both portable performance and correctness. Portable correctness is possible for serial codes thanks to IEEE 754 but is is challenging for parallel programs for reasons described in Chapter IV. Portable performance is challenging as architectures and compilers change. When analyzing the performance-accuracy tradeoff, the ultimate issue is one of numerical analysis, not quirks of the latest microarchitecture. This is why we focused on Newton-Ralphson iteration; gaining insight into which numerical routines are *fundamentally* more difficult is more useful in the long run than any performance experiment. This is not to say performance should be ignored; for any numerical code to be useful it must be executed. But instead, there is ample opportunity for the research community to blend numerical analysis with floating-point error analysis, both to analyze correctness and ensure performance portability.

CHAPTER VI

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

6.1 Summary

Verification of low-level problems is fundamentally hard. When dealing with verification, one butts up against undecidability because of Rice's Theorem (intuitively, Rice's theorem states any nontrivial property of a Turing machine is undecidable [157]). And further, the low-level implementation details one must care about when writing compilers, assembly languages, or highly-optimized routines by their very definition have little abstraction to aid in both programming and reasoning. To top it off, floating-point (FP) arithmetic can never be perfect because it is an approximation of an uncomputable set, the real numbers. These apparent limitations are why I'm so drawn to formal methods as a field and especially recent work in low-level verification techniques in binary analysis, static analysis, and FP arithmetic. Despite the difficulty, this effort is worth it: advances in formal methods have resulted in greater reliability of software and allowed lives to safely depend on computer programs behaving correctly. This dissertation gives programmers more high-level techniques to verify assembly languages and floating-point programs in order to write safe, correct, and optimized computer programs.

In Chapter III, we described a tool called Quameleon for compiler designers and low-level programmers to verify their code. Quameleon works by providing a domain-specific language for specifying instruction set architecture (ISA) semantics such that these semantics can both be used to check for bugs in the ISA specification itself as well be used as a binary analysis tool. We accomplished this by specifying an intermediate language, QIL, to which all other specifications convert. Quameleon is designed to make it easy to specify an ISAs semantics and then do generic verification tasks on binaries. This represents a rather specific use case: verification of binaries for obscure or old ISAs. Another place where verification research is needed is modern scientific computing applications. These applications rely on performance and correctness of numerical programs which, while in theory can be specified by Quameleon, are not well-suited for numerical and floating-point programs.

And so, the remainder of this dissertation focused on FP programs. We began in Chapter IV by looking at a one aspect of the MPI standard: reduction operations. MPI has been disparagingly referred to as the "assembly language of parallel programming" [177]. This is because the MPI standard itself operates at an unusual level of abstraction: it abstracts away the particular communication paradigms of the underlying networking hardware while also specifying things like data layout more concretely than the languages in which MPI routines are called (typically C, C++, or Fortran). Regardless, MPI is the *de facto* protocol for communication across nodes of a supercomputer.

From a numerical standpoint, the MPI standard obeys real-number semantics which do not hold for FP operations, namely associativity. Chapter IV looked at a statistical—as opposed to deterministic—analysis of MPI reduction errors. A statistical analysis makes sense here because the MPI standard *permits* nondeterminism for reduction operations. Chapter IV improved on existing random sampling of ordering by generating all possible random reduction trees. The main result was a method to generate realistic reduction trees for MPI programs and how such probabilistic error intervals were typically *smaller* than their serial implementations.

This discussion on error analysis of a simple reduction raises the next question: what's next? Adding a sequence of numbers can hardly be considered an "interesting" numerical program. Chapter V expanded analysis from a sequence of sums to more complex operations, starting with inner products and moving towards normalizing a vector. This chapter also moved from statistical analyses to deterministic analyses to ensure error bounds were sound. To better understand vector normalization, we looked at several optimized methods for computing reciprocal square root to show the tradeoff between performance and accuracy.

And so, this dissertation is another contribution towards verifying low-level computer programs. We provided techniques to correctly specify assembly language semantics (Chapter III), analyzed error of Message Passing Interface (MPI) reduction operations (Chapter IV), and analyzed the numerical properties of highly-optimized floating-point kernels (Chapter V). We end this dissertation by describing some open problems and potential strategies for solving these problems.

6.2 Future Research Directions

6.2.1 Binary Analysis. With more and more lives depending on correct computer programs, some researchers [105] are pushing forward the concept of "one Q.E.D." This ideal refers to a single proof of correctness that describes the entire behavior of a program along with its transformation between high-level language to binary and even to the hardware and operating system on which that binary is running. One part of this overall correctness proof is the translation between a compiler's program representation and machine code, described by ISAs.

In Chapter III we introduced a tool called Quameleon, a tool for binary analysis and ISA specification. Quameleon could be used as part of this "one Q.E.D." Quameleon could be extended to work for one part of this ambitious correctness proof, namely the semantics describing the binary which must be emitted by a verified compiler. Beyond this, Quameleon was designed to work on obscure and old architectures for which few other analysis tools exist. An exciting research direction is to work towards specifying these architectures by linking Quameleon with another binary analysis specification language called SAIL [5]. This strengthens Quameleon's verification argument because SAIL produces Coq theorems of correctness and these can be further proved in an automated way when possible and manually when necessary. This would require careful analysis of the features of Quameleon and SAIL to determine their interoperability.

6.2.2 The Emerging Field of Formal Numerical Methods. Perhaps the most important scientist of floating point, William Kahan, in 1983 began his treatise *Mathematics Written in Sand* with:

> Simplicity is a Virtue; yet we continue to cram ever more complicated circuits ever more densely into silicon chips, hoping all the while that their internal complexity will promote simplicity of use [108].

Kahan then went to indicate many examples where internal complexity resulted in outward complexity. This helped bring forth issues of FP correctness and shortly thereafter the IEEE 754 standard was finalized in 1985. It is partially because of the IEEE 754 standard that the issue of numerical program reproducibility can be mostly ignored for most programmers. But computing in 2021 is very different from 1983: CPUs back then had on the order of 100,000 transistors; now they have upwards of 40 billion. At the same time, computing (and along with it FP arithmetic) has become increasingly important in our daily lives. Simplicity of use has increased along with transistors, but only because of the hard work of many programmers and scientists. Despite this, FP arithmetic can be seen as somewhat unpredictable [81]; the underlying problem of calculating the error of a sequence of floating-point operations (FLOPs) is unsolved and creates a lack of trust in FP in general.

Perhaps serendipitously, the time of the ubiquity of IEEE 754 is ending. New floating-point standards such as bfloat16 [101] and posits [87] are causing programmers to look more closely once again at floating point. The community needs not only rigorous but also generalizable FP error analysis.

We wish to develop this field of formal numerical methods two different ways: probabilistic and deterministic. In Chapter IV we looked at probabilistic error analysis of MPI reduction routines. Even though FP error is deterministic, the MPI standard permits nondeterminism and so probabilistic error analysis can provide more realistic confidence intervals compared to deterministic error analysis. Conversely, in Chapter V we developed deterministic error analysis with the goal of maintaining soundness (i.e., never underapproximating true error). There remains work in extending this analysis to a more complete set of most important MPI operations and BLAS subroutines.

The drawback to probabilistic error analysis is it may be a poor model of actual code behavior (whence the logical fallacy "Everything I know nothing about is equally improbable" [109]). The drawback to deterministic error analysis is both its lack of scalability and sometimes unrealistically conservative bounds. But complete error analysis requires understanding the subtle interplay between FP error arising from nondeterministic sources (i.e., MPI and other parallel operations) and deterministic sources (i.e., individual FLOPs). HPC lags behind in the scientific rigor of other disciplines: there is no standardized way to describe error bounds for large-scale FP programs. We wish to allow the annotation of source code with error bounds and track these changes throughout program execution. This is currently done only for small research languages but could also be done for more realistic domain-specific frameworks such as PETSc [10]. To completely analyze such frameworks which produce large, parallel programs, we wish to unify the two approaches: deterministic error analysis for the serial portions of an FP program and probabilistic error analysis for the parallel portions. Through these, HPC can be seen as a more legitimate science by providing better evidence of program correctness.

6.2.3 Precomputation, Once Again. We end this dissertation with a small C function which captures its thesis: using high-level reasoning techniques to verify low-level computer programs. As we saw in Chapter V, the quality of an initial guess is of paramount importance for

Listing 6.1 An annotated version of the *fast reciprocal square root* function from the source code of *Quake III Arena*.

```
 1  float Q_rsqrt(float number) {
 2      long i;
 3      float x2, y;
 4      const float threehalfs = 1.5F;
 5      x2 = number * 0.5F;
 6      y  = number;
 7      // Reinterpret float as integer
 8      i  = *(long *) &y;
 9      // A good initial guess
10      i  = 0x5f3759df - (i >> 1);
11      y  = *(float *) &i;
12      // 1st iteration
13      y  = y * (threehalfs - (x2 * y * y));
14      // 2nd iteration, this can be removed
15      // y  = y * (threehalfs - (x2 * y * y));
16      return y;
17  }
```

Newton-Ralphson iteration. Therefore, for common mathematical operations it is worth the effort to hand-write assembly and undertake a detailed analysis of error. Early methods for calculating mathematical functions often used lookup tables to provide a good initial guess [182], but this requires a memory access. However, there are some situations where the calculation of an initial guess using a (relatively slow) lookup table can be replaced with (faster) integer operations on the bits of an FP number.

Among technology enthusiasts, the so-called *fast reciprocal square root* (`Q_rsqrt`) is one of the more famous "hacks" which gained popularity after being found in the source code for a popular computer game, *Quake III Arena*. However, this function had been around for much longer, which perhaps added to its notoriety. The main idea of `Q_rsqrt` is a fast, accurate initial guess followed by Newton-Ralphson iteration. We described such an iteration in (5.8). One reason for its popularity is its inscrutability, especially because of its "magic" constant `0x5f3759df`. We reproduce the function in Listing 6.1. The magic constant (Line 10) was precomputed using some algebraic properties of $1/\sqrt{x}$ and IEEE 754 single-precision numbers and resulted in a performance improvement over implementations using exponent lookup tables. We will not delve into the details of the details of `Q_rsqrt`; an excellent comprehensive explanation is given by Lomont [128].

However, we use `Q_rsqrt` as a good example of hand-optimized code which is not easily analyzable with existing floating-point error analysis tools. This does not mean work has not been done in this area: hardware verification engineers use robust verification techniques such as those presented by Russinoff [168] and bit-level manipulations on FP numbers have been verified by Stanford researchers [122]. The most difficult portion of Listing 6.1 from a verification standpoint is Line 10. To begin, it is not portable according to the C standard since it assumes a certain representation of the `float` type. Even ignoring this, it is not immediately clear the numerical effect of the right-shift operator in an FP number (it is not, like an integer, division by a power of two).

Libraries like Flocq define semantics of FP arithmetic in Coq [28] but these can be unwieldy and time-consuming libraries with which to prove numerical programs. Flocq is used in the gold standard of verified compilers, CompCert [125]. Unfortunately, for architectural reasons, extending this sort of bit-level analysis to FP numbers would require an architectural overhaul of CompCert. Even so, an interesting future research direction is allowing these types of operations in existing error analysis frameworks. For example, interpreting Line 10 is highly nontrivial. Worse still, the precise behavior of right-shift is implementation-defined (specifically whether right-shift is arithmetic or logical). That is, proofs of correctness would depend on architecture. Another interesting feature about `Q_rsqrt` is its resistance to generalization: this trick is no longer the optimal way to compute a reciprocal square root approximation. Moreover, while we may analyze a unary 32-bit FLOP exhaustively, that is about the limits of exhaustive checking. This means we need better tools to automatically analyze and synthesize such floating-point kernels. We hope to build more complete and formal models of processors, including both cost semantics and bit-level representations, to take advantage of tricks such as those used in `Q_rsqrt`.

Therefore, verification of such low-level programs is challenging. But such verification is an interesting research direction because computational kernels such as the reciprocal square root are known to be widely used. It is worth large amounts of human and computer time to optimize such routines. A commonly-held belief, reinforced by the fascination around `Q_rsqrt`, is that these routines have already been so highly optimized that no progress can be made. This is not true. Most recently, a guided exhaustive search forms the basis for RLibm, which is both better-performing *and* more accurate than any math library for single-precision FP numbers [127]. This work opens up opportunities for improvement of our most fundamental FP programs. Since exhaustive search is not tractable for 64-bit FP numbers or higher-arity functions, we predict future optimizations of mathematical functions will require not only the computational power of future computer architectures but also more powerful models of floating-point error.

This resurgence of effort optimizing our most fundamental numerical programs is reminiscent of the early days of computing when computer cycles, memory, and applying software updates were much more expensive than they are today. Back then, it was economical to not only spend more human time up-front to carefully verify computer programs, but also spend computer cycles to precompute (sometimes called partially evaluate) wherever possible. Now, computer cycles are cheap but the paradigm is similar: the sheer number of computers used throughout the world, especially those in low-power and mission-critical applications, make the effort of verification and optimization worthwhile. And so, as computers become even more ubiquitous, improvements to correctness and performance can have a large effect on computing worldwide.

APPENDIX A

PROOFS AND REPRODUCIBILITY FOR CHAPTER IV

A.1 Proof of (4.2)

We proceed by proving the limit of the ratios diverge. Observe:

$$\lim_{n\to\infty} \frac{C_n}{n!/2} = \lim_{n\to\infty} \frac{\frac{(2n)!}{(n+1)!n!}}{n!/2} = \lim_{n\to\infty} \frac{2(2n)!}{(n+1)!} = \infty.$$

Next, we find it more convenient to use $n+1$ instead of n for the second inequality. So

$$\lim_{n\to\infty} \frac{g_{n+1}}{(n+1)!/2} = \lim_{n\to\infty} \frac{2(2n-1)!!}{(n+1)!}.$$

Since $2n-1$ is odd, we have

$$(2n-1)!! = \frac{(2n-1)!}{(2(n-1))!!}.$$

And further, since $2(n-1)$ is even, we can factor out a 2 from each term of $(2(n-1))!!$:

$$(2n-1)!! = \frac{(2n-1)!}{(2(n-1))!!} = \frac{(2n-1)!}{2^{n-1}(n-1)!}.$$

Putting this all together,

$$\begin{aligned}\lim_{n\to\infty} \frac{2(2n-1)!!}{(n+1)!} &= \lim_{n\to\infty} \frac{\frac{2(2n-1)!}{2^{n-1}(n-1)!}}{(n+1)!}\\ &= \lim_{n\to\infty} \frac{\frac{(2n-1)!}{2^{n}(n-1)!}}{(n+1)n(n-1)!}\\ &= \lim_{n\to\infty} \frac{(2n-1)!}{2^n(n+1)n} = \infty\end{aligned}$$

because the factorial function grows faster than 2^n. □

A.2 Reproducibility

All software used to generate the figures and results from this chapter are available under the GNU GPLv3 license at `https://github.com/sampollard/reduce-error` using the tag `correctness-2020`. The data used to generate the results in this paper are available under the GNU GPLv3 license at `https://dx.doi.org/10.5281/zenodo.4047699`. Instructions to re-generate the datasets are included in the `README.md` provided in either repository.

Experiments were performed on two Linux nodes:

1. One node running Ubuntu 18.04.4 LTS on Linux `4.15.0-55-generic`, two Intel Xeon Gold 6148 CPUs at 2.40GHz, and 384GiB DDR4 RAM at 2666MHz.

2. Single-node provisions from the University of Oregon's local cluster, Talapas, running Linux `3.10.0-957.27.2.el7.x86_64`, two Intel Xeon E5-2690v4 CPUs at 2.60Ghz, 128GiB DDR4 RAM, IBM GPFS file system.

Software dependencies are as follows.

- Lmod environment modules 7.7 to manage environment variables.
- Shell scripts were run using Bash 4.4.20.
- Data analysis and figure generation were performed with `R` 4.0.2 and ggplot2 3.3.2.
- Spack 0.13.4 to manage Boost and MPFR
- Boost 1.72.0
  * Boost was loaded with

    `module load boost-1.72.0-gcc-7.5.0-q725eoa`
- Simgrid 3.25.1, commit hash `4b7251c4ac80`
- GNU MPFR 4.0.2
  * MPFR was loaded using

    `module load mpfr-4.0.2-gcc-7.5.0-fveqzlf`
- GCC and OpenMPI
  * GCC 7.5.0 and OpenMPI 4.0.3 on node 1.
  * GCC 7.3.0 and OpenMPI 2.1 on node 2.

APPENDIX B

REPRODUCIBILITY FOR CHAPTER V

The software used to generate figures and results from this chapter are available under the GNU GPLv3 license at `https://github.com/sampollard/fpcost` in the branch `dissertation`. The data and source code used to generate the results in this paper are available under the GNU GPLv3 license at `https://doi.org/10.5281/zenodo.4745336`.

Experiments were performed on one Linux node and one laptop:

1. One node running Ubuntu 18.04.5 LTS, Linux `5.3.0-70-generic`, two Intel Xeon E5-2699 v3 CPUs at 2.30GHz, and 256GiB DDR4 RAM at 2133MHz
2. A laptop running Ubuntu 20.04.2 LTS, Linux `5.8.0-50-generic`, one Intel Core i7-10610U CPU at 1.80GHz, and 16GiB DDR4 RAM at 2667MHz.

Software dependencies are as follows.

- Data analysis and figure generation were performed with `R` 3.4.4 with ggplot2 3.3.3.
- GNU MPFR 4.0.1-1
- FPTaylor `https://github.com/soarlab/FPTaylor` commit hash `fabc895ddc595dc41fbe7ce511049c745e1f9136`
- Ocaml 4.0.5
- Satire `https://github.com/arnabd88/Satire` commit hash `ee69bb7eaed51e96af727f2690aad9a414fb4512`
  * Python 3.8.5
  * Rust 1.47.0
  * Gelpia `https://github.com/soarlab/gelpia` commit hash `c28bf25593423f71ce6ef86122f2a8aa22bf0b33`

REFERENCES CITED

[1] Affeldt, R., and Marti, N. An approach to formal verification of arithmetic functions in assembly. In *11th Asian Computing Science Conference (ASIAN)* (Tokyo, Japan, Dec. 2006), M. Okada and I. Satoh, Eds., Advances in Computer Science: Secure Software and Related Issues (LNCS 4435), Springer-Verlag, pp. 346–360.

[2] Amla, N., Banerjee, A., Cosley, D. R., Ding, W., Mutka, M. W., Roy, S., and Sprintson, A. Formal methods in the field (FMitF): Program solicitation. Tech. rep., National Science Foundation, Alexandria, VA, USA, Feb. 2022.

[3] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. v3.9.1 released April 1, 2021.

[4] Appel, A. W. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems* (Saarbrücken, Germany, Mar. 2011), ESOP/ETAPS (LNCS 6602), Springer-Verlag, pp. 1–17.

[5] Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K. E., Norton, R. M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., and Sewell, P. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages 3*, POPL (Jan. 2019), 71:1–71:31.

[6] Armstrong, R. C., Punnoose, R. J., Wong, M. H., and Mayo, J. R. Survey of existing tools for formal verification. Tech. rep., Sandia National Laboratories, Albuquerque, NM, USA, Dec. 2014.

[7] Arteaga, A., Fuhrer, O., and Hoefler, T. Designing bit-reproducible portable high-performance applications. In *28th International Parallel and Distributed Processing Symposium* (2014), IPDPS, IEEE, pp. 1235–1244.

[8] Ayewah, N., Hovermeyer, D., Morgenthaler, J. D., Penix, J., and Pugh, W. Using static analysis to find bugs. *IEEE Software 25*, 5 (Sept. 2008), 22–29.

[9] Balaji, P., and Kimpe, D. On the reproducibility of MPI reduction operations. In *10th International Conference on High Performance Computing and Communications & International Conference on Embedded and Ubiquitous Computing* (Zhangjiajie, China, 2013), HPCC/EUC, IEEE, pp. 407–414.

[10] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.15, Argonne National Laboratory, 2021.

[11] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys 51*, 3 (July 2018), 50:1–50:39.

[12] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, UT, USA, June 2001), PLDI '01, ACM, pp. 203–213.

[13] Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[14] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects* (Amsterdam, Netherlands, Nov. 2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds., FMCO (LNCS 4111), Springer-Verlag, pp. 364–387.

[15] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. Cvc4. In *23rd International Conference on Computer Aided Verification* (Snowbird, UT, USA, 2011), G. Gopalakrishnan and S. Qadeer, Eds., CAV (LNCS 6806), Springer-Verlag, pp. 171–177.

[16] Barrett, C., Fontaine, P., and Tinelli, C. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories* (Edinburgh, United Kingdom, July 2010), CAV/SAT 2010, pp. 1–14.

[17] Barrett, C., and Tinelli, C. Satisfiability modulo theories. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer-Verlag, Basel, Switzerland, 2018, pp. 305–343.

[18] Barrett, G. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering 15*, 5 (May 1989), 611–621.

[19] Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M. O., and Fox, A. A verified certificate checker for finite-precision error bounds in coq and hol4. In *Formal Methods in Computer Aided Design (FMCAD)* (2018), pp. 1–10.

[20] Bhadra, J., Abadir, M. S., Wang, L.-C., and Ray, S. A survey of hybrid techniques for functional verification. *IEEE Design Test of Computers 24*, 2 (Mar. 2007), 112–122.

[21] Bindel, D., Demmel, J., Kahan, W., and Marques, O. On computing givens rotations reliably and efficiently. *ACM Transactions on Mathematical Software 28*, 2 (June 2002), 206–238.

[22] Blackford, S. Lapack users' guide: Accuracy and stability. Tech. rep., University of Tennessee, Aug. 1999. Available at `https://www.netlib.org/lapack/lug/node72.html`.

[23] Blackford, S. L., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., and Whaley, R. C. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathemathical Software 28*, 2 (June 2002), 135–151.

[24] Blair, M., Obenski, S., and Bridickas, P. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Tech. Rep. IMTEC-92-26, U.S. Government Accountability Office, Feb. 1992.

[25] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*. LNCS 2566. Springer-Verlag, Berlin, Germany, 2002, pp. 85–108.

[26] Blom, S., Darabi, S., and Huisman, M. Verification of loop parallelisations. In *Fundamental Approaches to Software Engineering* (London, United Kingdom, 2015), A. Egyed and I. Schaefer, Eds., FACE (LNCS 9033), Springer-Verlag, pp. 202–217.

[27] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd your herd of provers. In *23rd International Conference on Automated Deduction* (Wroclaw, Poland, 2011), First International Workshop on Intermediate Verification Languages: Boogie, pp. 53–64.

[28] Boldo, S., and Melquiond, G. Flocq: A unified library for proving floating-point algorithms in coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic* (Tübingen, Germany, July 2011), E. Antelo, D. Hough, and P. Ienne, Eds., ARITH '11, IEEE Computer Society, pp. 243–252.

[29] Boldo, S., and Melquiond, G. *Computer Arithmetic and Formal Proofs: Verifying Floating-Point Algorithms with the Coq System*, 1st ed. ISTE Press - Elsevier, United Kingdom, Nov. 2017.

[30] Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, CA, USA, Apr. 1975), ACM, pp. 234–245.

[31] Brain, M., D'Silva, V., Griggio, A., Haller, L., and Kroening, D. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design 45*, 2 (Oct. 2014), 213–245.

[32] Brain, M., Tinelli, C., Ruemmer, P., and Wahl, T. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *22nd Symposium on Computer Arithmetic* (Lyon, France, June 2015), ARITH, pp. 160–167.

[33] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. BAP: A binary analysis platform. In *Computer Aided Verification (CAV)* (Snowbird, UT, USA, July 2011), G. Gopalakrishnan and S. Qadeer, Eds., CAV (LNCS 6806), Springer-Verlag, pp. 463–469.

[34] Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. The OpenSMT solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Paphos, Cyprus, Mar. 2010), TACAS '10, Springer-Verlag, pp. 150–153.

[35] Buchwald, S. Optgen: A generator for local optimizations. In *Proceedings of the 24th International Conference on Compiler Construction* (Apr. 2015), B. Franke, Ed., CC (LNCS 9031), Springer-Verlag, pp. 171–189.

[36] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, CA, USA, Dec. 2008), OSDI '08, USENIX Association, pp. 209–224.

[37] 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.

[38] 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.

[39] Chiang, W.-F., Gopalakrishnan, G., Rakamaric, Z., and Solovyev, A. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2014), PPoPP '14, ACM, pp. 43–52.

[40] Church, A. A formulation of the simple theory of types. *The Journal of Symbolic Logic 5*, 2 (1940), 56–68.

[41] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification* (Copenhagen, Denmark, July 2002), E. Brinksma and K. G. Larsen, Eds., CAV (LNCS 2404), Springer-Verlag, pp. 359–364.

[42] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (Montreal, Canada, Sept. 2000), ICFP '00, ACM, pp. 268–279.

[43] Clarke, L., Glendinning, I., and Hempel, R. The MPI message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems* (Basel, 1994), K. M. Decker and R. M. Rehmann, Eds., Birkhäuser Basel, pp. 213–218.

[44] Clarke, L. A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering 2*, 3 (Sept. 1976), 215–222.

[45] Collingbourne, P., Cadar, C., and Kelly, P. H. J. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria, Apr. 2011), EuroSys '11, ACM, pp. 315–328.

[46] Conchon, S., Contejean, E., Kanig, J., and Lescuyer, S. Lightweight integration of the ergo theorem prover inside a proof assistant. In *Proceedings of the Second Workshop on Automated Formal Methods* (Atlanta, GA, USA, 2007), AFM '07, ACM, pp. 55–59.

[47] Conchon, S., Iguernelala, M., Ji, K., Melquiond, G., and Fumex, C. A three-tier strategy for reasoning about floating-point numbers in smt. In *29th International Conference on Computer Aided Verification* (Heidelberg, Germany, July 2017), V. Kuncak and R. Majumdar, Eds., CAV (LNCS 10426, 10427), Springer-Verlag, pp. 419–435.

[48] Conchon, S., Melquiond, G., Roux, C., and Iguernelala, M. Built-in treatment of an axiomatic floating-point theory for smt solvers. In *10th International Workshop on Satisfiability Modulo Theories* (Manchester, United Kingdom, June 2012), P. Fontaine and A. Goel, Eds., SMT '12, pp. 12–21.

[49] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation 76*, 2 (1988), 95–120.

[50] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, CA, USA, Jan. 1977), POPL '77, ACM, pp. 238–252.

[51] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTREÉ analyzer. In *European Symposium on Programming Languages and Systems* (Edinburgh, United Kingdom, Apr. 2005), M. Sagiv, Ed., ESOP (LNCS 3444), Springer-Verlag, pp. 21–30.

[52] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c. In *Software Engineering and Formal Methods* (Thessaloniki, Greece, Oct. 2012), G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds., SFEM (LNCS 7504), Springer-Verlag, pp. 233–247.

[53] Dale, M., and Moon, J. The permuted analogues of three catalan sets. *Journal of statistical planning and inference 34*, 1 (Jan. 1993), 75–87.

[54] Daniel, J. W., Gragg, W. B., Kaufman, L., and Stewart, G. W. Reorthogonalization and stable algorithms for updating the gram-schmidt qr factorization. *Mathematics of Computation 30*, 136 (1976), 772–795.

[55] Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., and Bastian, R. Daisy - framework for analysis and optimization of numerical programs (tool paper). In *Tools and Algorithms for the Construction and Analysis of Systems* (Thessaloniki, Greece, 2018), D. Beyer and M. Huisman, Eds., Springer International Publishing, pp. 270–287.

[56] Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S., and Panchekha, P. Scalable yet rigorous floating-point error analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), SC '20, IEEE Press, pp. 51:1–51:14.

[57] Daumas, M., Rideau, L., and Théry, L. A generic library for floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics* (Edinburgh, United Kingdom, Sept. 2001), R. J. Boulton and P. B. Jackson, Eds., TPHOLs (LNCS 2152), Springer-Verlag, pp. 169–184.

[58] de Dinechin, F., Defour, D., and Lauter, C. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research Report RR-5137, LIP RR-2004-10, INRIA, LIP, 2004.

[59] de Dinechin, F., Lauter, C., and Melquiond, G. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers 60*, 2 (Feb. 2011), 242–253.

[60] de Dinechin, F., Lauter, C. Q., and Melquiond, G. Assisted verification of elementary functions using gappa. In *Proceedings of the ACM Symposium on Applied Computing* (Dijon, France, 2006), SAC '06, ACM, pp. 1318–1322.

[61] de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In *International Conference on Automated Deduction* (Berlin, Germany, Aug. 2015), CADE-25, Springer-Verlag, pp. 378–388.

[62] de Moura, L. M., and Bjørner, N. Proofs and refutations, and Z3. In *7th International Workshop on the Implementation of Logics at the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (Doha, Qatar, Nov. 2008), IWIL LPAR 2008.

[63] Demeure, N. *Compromise Between Precision and Performance in High Performance Computing*. Theses, École Normale supérieure Paris-Saclay, Jan. 2021.

[64] Demmel, J., Ahrens, P., and Nguyen, H. D. Efficient reproducible floating point summation and BLAS. Tech. Rep. UCB/EECS-2016-121, EECS Department, University of California, Berkeley, June 2016.

[65] Demmel, J., and Nguyen, H. D. Parallel reproducible summation. *IEEE Transactions on Computers 64*, 7 (2015), 2060–2070.

[66] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (Aug. 1975), 453–457.

[67] Dongarra, J., Hammarling, S., Higham, N. J., Relton, S. D., Valero-Lara, P., and Zounon, M. The design and performance of batched blas on modern high-performance computing systems. In *International Conference on Computational Science* (Zurich, Switzerland, 2017), vol. 108 of *ICCS*, pp. 495–504.

[68] Edelman, A. The mathematics of the pentium division bug. *SIAM Review 39*, 1 (1997), 54–67.

[69] Espelid, T. O. On floating-point summation. *SIAM Review 37*, 4 (1995), 603–607.

[70] Evangelista, S. High level petri nets analysis with helena. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets* (Miami, FL, USA, June 2005), ICATPN '05, Springer-Verlag, pp. 455–464.

[71] Facebook Open Source Community. Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships. Available at `https://fbinfer.com`, 2019.

[72] Filliâtre, J.-C., and Paskevich, A. Why3 — where programs meet provers. In *22nd European Symposium on Programming: Programming Languages and Systems* (Rome, Italy, Mar. 2013), M. Felleisen and P. Gardner, Eds., ESOP (LNCS 7792), Springer-Verlag, pp. 125–128.

[73] Fischer, P., and Heisey, K. NEKBONE: Thermal hydraulics mini-application. Tech. rep., Argonne National Laboratory, Lemont, IL, United States, 2013. Available at `https://github.com/Nek5000/Nekbone`.

[74] Fischer, P., and Kerkemeier, S. NEK: a fast and scalable high-order solver for computational fluid dynamics, 2020. Available at `https://nek5000.mcs.anl.gov/`.

[75] Flanagan, C., and Qadeer, S. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, OR, USA, 2002), POPL '02, ACM, pp. 191–202.

[76] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. Mpfr: A multiple-precision binary floating-point library with correct rounding. *Transactions on Mathematical Software 33*, 2 (June 2007).

[77] Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., and Swamy, N. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages 3*, POPL (Jan. 2019), 63:1–63:30.

[78] Fujitsu Limited. *A64FX Microarchitecture Manual, Version 1.4*. Kawasaki, Japan, Mar. 2021. Available at `https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.4.pdf`.

[79] Galassi, M., Davies, J., Theiler, J., Gough, B., and Jungman, G. *GNU Scientific Library - Reference Manual, for GSL Version 1.12*, 3rd ed. Free Software Foundation, Jan. 2009. Software v2.6. Available at `https://www.gnu.org/software/gsl/`.

[80] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*, 1 (Mar. 1991), 5–48.

[81] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*, 1 (Mar. 1991), 5–48.

[82] Gopalakrishnan, G., Hovland, P. D., Iancu, C., Krishnamoorthy, S., Laguna, I., Lethin, R. A., Sen, K., Siegel, S. F., and Solar-Lezama, A. Report of the HPC correctness summit, jan 25-26, 2017, washington DC. Tech. rep., Department of Energy, 2017. Available at `https://arxiv.org/abs/1705.07478`.

[83] Gordon, M. J. C. *VLSI Specification, Verification and Synthesis*, vol. 35 of *The Kluwer International Series in Engineering and Computer Science*. Springer-Verlag, Boston, MA, USA, 1988, ch. HOL: A Proof Generating System for Higher-Order Logic, pp. 73–128.

[84] Goualard, F. How do you compute the midpoint of an interval? *ACM Transactions on Mathematical Software 40*, 2 (Mar. 2014), 11:1–11:25.

[85] Goubault, E., and Putot, S. Static analysis of finite precision computations. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation* (Austin, TX, USA, 2011), R. Jhala and D. Schmidt, Eds., VMCAI (LNCS 6538), Springer-Verlag, pp. 232–247.

[86] Graf, S., and Saidi, H. Construction of abstract state graphs with PVS. In *Computer Aided Verification* (Haifa, Israel, June 1997), O. Grumberg, Ed., CAV (LNCS 1254), Springer-Verlag, pp. 72–83.

[87] Gustafson, J., and Yonemoto, I. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations 4*, 2 (2017), 71–86.

[88] Hales, T. C. Formal proof. *Notices of the AMS 55*, 11 (2008), 1370–1380.

[89] Harrison, J. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification* (Bertinoro, Italy, May 2006), M. Bernardo and A. Cimatti, Eds., SFM (LNCS 3965), Springer-Verlag, pp. 211–242.

[90] Harrison, J. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge, United Kingdom, 2009.

[91] Hawblitzel, C., Petrank, E., Qadeer, S., and Tasiran, S. Automated and modular refinement reasoning for concurrent programs. In *27th International Conference on Computer Aided Verification* (San Francisco, CA, USA, July 2015), D. Kroening and C. S. Păsăreanu, Eds., CAV (LNCS 9207), Springer-Verlag, pp. 449–465.

[92] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, OR, USA, 2002), POPL '02, ACM, pp. 58–70.

[93] Hex-Rays. The IDA disassembler and debugger, 2018. Available at https://www.hex-rays.com.

[94] Higham, N. J. The accuracy of floating point summation. *SIAM Journal of Scientific Computing 14*, 4 (1993), 783–799.

[95] Higham, N. J. *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[96] Higham, N. J. *Accuracy and Stability of Numerical Algorithms*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, United States, 2002.

[97] Higham, N. J., and Mary, T. A new approach to probabilistic rounding error analysis. *SIAM Journal on Scientific Computing 41*, 5 (2019), A2815–A2835.

[98] Hoffeins, F., Ciorba, F. M., and Banicescu, I. Examining the reproducibility of using dynamic loop scheduling techniques in scientific applications. In *IEEE International Parallel and Distributed Processing Symposium Workshops* (Lake Buena Vista, FL, USA, 2017), IPDPSW, IEEE, pp. 1579–1587.

[99] Holzman, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–295.

[100] INRIA Saclay, Île-de-France. *Toccata: Formally Verified Programs, Certified Tools and Numerical Computations*. Website at `http://toccata.lri.fr/fp.en.html`.

[101] Intel Corporation. BFLOAT16 — hardware numerics definition. Tech. Rep. 338302-001US, Nov. 2018.

[102] Intel Corporation. *Intel Intrinsics Guide*, 3.5.4 ed., Oct. 2020. Available at `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[103] Ipsen, I. C. F., and Zhou, H. Probabilistic error analysis for inner products. *SIAM Journal on Matrix Analysis and Applications 41*, 4 (June 2020), 1726–1741.

[104] Johnson-Freyd, P. A. Refinement and composition in formal modeling of temporal systems. Available at `http://www.cs.uoregon.edu/Reports/AREA-201511-Johnson-Freyd.pdf`, 2015. Area Exam.

[105] Johnson-Freyd, P. A., 2019. Internal Discussion at Sandia National Laboratories.

[106] Jung, M., Kim, S., Han, H., Choi, J., and Cha, S. K. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research* (2019).

[107] Kahan, W. Pracniques: Further remarks on reducing truncation errors. *Communications of the ACM 8*, 1 (Jan. 1965), 40.

[108] Kahan, W. Mathematics written in sand - the hp-15C, intel 8087, etc. In *Proceedings of the American Statistical Association* (1983), Joint Statistical Meeting, pp. 12–26. Available at `https://people.eecs.berkeley.edu/~wkahan/MathSand.pdf`.

[109] Kahan, W. The improbability of probabilistic error analysis for numerical computations. UCB Statistics Colloquium, 1998. Available at `https://people.eecs.berkeley.edu/~wkahan/improber.pdf`.

[110] Kaplan, D. M. Correctness of a compiler for algol-like programs. Tech. rep., Stanford University, 1967. Stanford Artificial Intelligence Memo No. 48.

[111] Keller, R. M. Formal verification of parallel programs. *Communications of the ACM 19*, 7 (July 1976), 371–384.

[112] Kneuper, R. Limits of formal methods. *Formal Aspects of Computing 9*, 4 (July 1997), 379–394.

[113] 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.

[114] Kulisch, U. Very fast and exact accumulation of products. *Computing 91*, 4 (Apr. 2011), 397–405.

[115] Laboratoire de Recherche en Informatique and Inria Saclay Ile-de-France. The alt-ergo automated theorem prover, version 2.2.0, Apr. 2006–2018. `https://alt-ergo.ocamlpro.com`.

[116] Laguna, I., and Rubio-González, C., Eds. *International Workshop on Software Correctness for HPC Applications* (New York, NY, USA, Nov. 2020), ACM.

[117] Laguna, I., Wood, P. C., Singh, R., and Bagchi, S. Gpumixer: Performance-driven floating-point tuning for gpu scientific applications. In *International Conference on High Performance Computing (ISC)* (Cham, Switzerland, 2019), M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., Springer International Publishing, pp. 227–246.

[118] Lakhotia, K., Tillmann, N., Harman, M., and de Halleux, J. FloPSy — search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems* (Natal, Brazil, Nov. 2010), A. Petrenko, A. Simão, and J. C. Maldonado, Eds., ICTSS (LNCS 6435), Springer-Verlag, pp. 142–157.

[119] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[120] Langlois, P., Martel, M., and Thévenoux, L. Accuracy versus time: A case study with summation algorithms. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation* (New York, NY, USA, 2010), PASCO '10, ACM, p. 121–130.

[121] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, CA, USA, Mar. 2004), CGO '04, IEEE Computer Society, pp. 75–.

[122] Lee, W., Sharma, R., and Aiken, A. On automatically proving the correctness of math.h implementations. *Proceedings of the ACM Programming Languages 2*, POPL (Dec. 2017).

[123] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal, Apr. 2010), E. M. Clarke and A. Voronkov, Eds., LPAR (LNCS 6355), Springer-Verlag, pp. 348–370.

[124] Lensink, L., Smetsers, S., and van Eekelen, M. Generating verifiable java code from verified pvs specifications. In *Proceedings of the 4th International Conference on NASA Formal Methods* (Norfolk, VA, USA, Apr. 2012), NFM'12, Springer-Verlag, pp. 310–325.

[125] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[126] Leroy, X. In search of software perfection. Available at `https://youtu.be/lAU5hx_3xRc`, Nov. 2016.

[127] Lim, J. P., Aanjaneya, M., Gustafson, J., and Nagarakatte, S. An approach to generate correctly rounded math libraries for new floating point variants. *Proceedings of the ACM on Programming Languages 5*, POPL (Jan. 2021), 29:1–29:30.

[128] Lomont, C. Fast inverse square root, 2003. Available at `http://lomont.org/papers/2003/InvSqrt.pdf`.

[129] Lopes, N. P., Menendez, D., Nagarakatte, S., and Regehr, J. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA, 2015), PLDI '15, ACM, pp. 22–32.

[130] Luu, D. Static v. dynamic languages literature review, Nov. 2014. Available at `https://danluu.com/empirical-pl/`.

[131] Marjamäki, D. Cppcheck: A tool for static c/c++ code analysis. `http://cppcheck.sourceforge.net/`, 2021. Accessed 6 Apr 2021.

[132] Marques-Silva, J., Lynce, I., and Malik, S. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsch, Eds. IOS Press, Amsterdam, Netherlands, 2008, ch. 4, pp. 127–148.

[133] Marsaglia, G., and Zaman, A. A new class of random number generators. *The Annals of Applied Probability 1*, 3 (1991), 462–480.

[134] Martel, M. Propagation of roundoff errors in finite precision computations: A semantics approach. In *Programming Languages and Systems* (Berlin, Germany, 2002), D. Le Métayer, Ed., Springer-Verlag, pp. 194–208.

[135] Matoussi, O., Durand, Y., Sentieys, O., and Molnos, A. Error analysis of the square root operation for the purpose of precision tuning: A case study on k-means. In *IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2019), vol. 2160-052X, pp. 75–82.

[136] Message Passing Interface Forum. MPI: A message-passing interface standard: Version 3.1. Tech. rep., MPI Forum, Knoxville, TN, United States, 2015.

[137] Michel, C., Rueher, M., and Lebbah, Y. Solving constraints over floating-point numbers. In *Principles and Practice of Constraint Programming* (Paphos, Cyprus, 2001), T. Walsh, Ed., CP (LNCS 2239), Springer-Verlag, pp. 524–538.

[138] Mittal, S. A survey of techniques for approximate computing. *ACM Computing Surveys 48*, 4 (Mar. 2016).

[139] Moore, J. S. A mechanically verified language implementation. *Journal of Automated Reasoning 5*, 4 (Dec. 1989), 461–492.

[140] Mullen, E., Zuniga, D., Tatlock, Z., and Grossman, D. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA, June 2016), PLDI '16, ACM, pp. 448–461.

[141] Muller, J.-M., Brisebarre, N., De Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., and Torres, S. *Handbook of Floating-Point Arithmetic*, second ed. Springer International Publishing AG, 2018.

[142] Muñoz, C. A. Formal methods in air traffic management: The case of unmanned aircraft systems (invited lecture). In *International Colloquium on Theoretical Aspects of Computing* (Cham, Switzerland, 2015), M. Leucker, C. Rueda, and F. D. Valencia, Eds., ITAC, Springer International Publishing, pp. 58–62.

[143] National Security Agency Research Directorate. Ghidra: A software reverse engineering (sre) framework, 2019. Available at `https://www.ghidra-sre.org`.

[144] Necula, G. C. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Vancouver, BC, Canada, 2000), PLDI '00, ACM, pp. 83–94.

[145] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, USA, June 2007), PLDI '07, ACM, pp. 89–100.

[146] Neumann, P. Mariner i — no holds BARred. *Forum on Risks to the Public in Computers and Related Systems 8*, 75 (May 1989).

[147] Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Germany, 2002.

[148] Patki, T., Thiagarajan, J. J., Ayala, A., and Islam, T. Z. Performance optimality or reproducibility: That is the question. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2019), SC '19, Association for Computing Machinery.

[149] Perconti, J. T., and Ahmed, A. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming Languages and Systems* (Grenoble, France, Apr. 2014), Z. Shao, Ed., ESOP (LNCS 8410), Springer-Verlag, pp. 128–148.

[150] Petri, C. A., and Reisig, W. Petri net. *Scholarpedia 3*, 4 (2008), 6477. revision #91647.

[151] 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.

[152] 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.

[153] Press, W. H., Teukolsky, Saul A. Vetterling, W. T., and Flannery, B. P. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, Cambridge, United Kingdom, 2007.

[154] Ray, B., Posnett, D., Filkov, V., and Devanbu, P. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China, Nov. 2014), FSE 2014, ACM, pp. 155–165.

[155] Reid, A. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design* (Mountain View, CA, USA, Oct. 2016), FMCAD, FMCAD Inc, pp. 161–168.

[156] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Copenhagen, Denmark, July 2002), LICS '02, IEEE Computer Society, pp. 55–74.

[157] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society 74* (1953).

[158] Rival, X., and Yi, K. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, Cambridge, MA, USA, Feb. 2020.

[159] Robertazzi, T. G., and Schwartz, S. C. Best "ordering" for floating-point addition. *Transactions on Mathematical Software 14*, 1 (Mar. 1988), 101–110.

[160] Rojas, R. Konrad zuse's legacy: The architecture of the Z1 and Z3. *Annals of the History of Computing 19*, 2 (1997), 5–16.

[161] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, CA, USA, Jan. 1988), POPL '88, pp. 12–27.

[162] Roşu, G., and Şerbănuţă, T. F. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming 79*, 6 (2010), 397–434.

[163] Rouhani, B., Lo, D., Zhao, R., Liu, M., Fowers, J., Ovtcharov, K., Vinogradsky, A., Massengill, S., Yang, L., Bittner, R., Forin, A., Zhu, H., Na, T., Patel, P., Che, S., Koppaka, L. C., Song, X., Som, S., Das, K., Tiwary, S., Reinhardt, S., Lanka, S., Chung, E., and Burger, D. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. In *Conference on Neural Information Processing Systems* (Nov. 2020), NeurIPS, ACM.

[164] Rubio-González, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 27:1–27:12.

[165] Rümmer, P., and Wahl, T. An smt-lib theory of binary floating-point arithmetic. In *8th International Workshop on Satisfiability Modulo Theories (SMT)* (Edinburgh, United Kingdom, July 2010), pp. 151–165.

[166] Rump, S. M. Ultimately fast accurate summation. *SIAM Journal of Scientific Computing 31*, 5 (2009), 3466–3502.

[167] Russell, S. Unifying logic and probability. *Communications of the ACM 58*, 7 (July 2015), 88–97.

[168] Russinoff, D. M. *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer Nature Switzerland AG, Cham, 2019.

[169] Russinoff, D. M., and Flatau, A. Mechanical verification of register-transfer logic: A floating-point multiplier. In *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J. S. Moore, Eds. Kluwer Academic Publishers, Dordrecht, Netherlands, June 2000.

[170] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at google. *Communications of the ACM 61*, 4 (Mar. 2018), 58–66.

[171] Sauer, T. *Numerical Analysis*, 3rd ed. Pearson, Boston, MA, USA, 2018.

[172] Schordan, M., and Quinlan, D. A source-to-source architecture for user-defined optimizations. In *Modular Programming Languages* (Klagenfurt, Austria, Aug. 2003), L. Böszörményi and P. Schojer, Eds., JMLC (LNCS 2789), Springer-Verlag, pp. 214–223.

[173] Shapiro, S., and Kouri Kissel, T. Classical logic. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., spring 2018 ed. Metaphysics Research Lab, Stanford University, 2018.

[174] Sharma, A., Paliwal, K. K., Imoto, S., and Miyano, S. Principal component analysis using qr decomposition. *International Journal of Machine Learning and Cybernetics 4*, 6 (2013), 679–683.

[175] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *37th IEEE Symposium on Security and Privacy* (San Jose, CA, USA, May 2016), SP, IEEE Computer Society.

[176] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 138–157.

[177] 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`.

[178] Solovyev, A., Jacobsen, C., Rakamarić, Z., and Gopalakrishnan, G. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *International Symposium on Formal Methods* (Oslo, Norway, 2015), N. Bjørner and F. de Boer, Eds., FM 2015, Springer International Publishing, pp. 532–550.

[179] Sørensen, M. H., and Urzyczyn, P. *Lectures on the Curry-Howard Isomorphism*, vol. 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, July 2006.

[180] Spivey, J. M. *Understanding Z: a Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, United Kingdom, 1988.

[181] Strakoš, Z., and Tichý, P. On error estimation in the conjugate gradient method and why it works in finite precision computations. *Electronic Transactions on Numerical Analysis 13* (2002), 56–80.

[182] Sun Microsystems. Correctly rounded square root: esqrt.c, 1995. Available at `http://www.netlib.org/fdlibm/`.

[183] Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., and Yang, J. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan, Sept. 2011), M. M. T. Chakravarty, Z. Hu, and O. Danvy, Eds., ACM, pp. 266–278.

[184] Tagliavini, G., Mach, S., Rossi, D., Marongiu, A., and Benin, L. A transprecision floating-point platform for ultra-low power computing. In *Design, Automation Test in Europe Conference Exhibition* (Dresden, Germany, Mar. 2018), DATE, pp. 1051–1056.

[185] Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications 19*, 1 (Feb. 2005), 49—–66.

[186] The Coq Development Team. The coq proof assistant, version 8.8.0, Apr. 2018.

[187] The Coq Development Team. *The Coq Reference Manual, version 8.9*, Jan. 2019. Available electronically at `http://coq.inria.fr/doc`.

[188] The Free Software Foundation. The GNU C library manual. Tech. rep., The Free Software Foundation, Boston, MA, United States, Aug. 2018.

[189] The Open MPI Development Team. *MPIReduce(3) man page*, 4.0.4 ed. Software in the Public Interest, June 2020.

[190] The SimGrid Team. SMPI: Simulate MPI applications, 2020. Available at `https://simgrid.org/doc/latest/app_smpi.html#mpi-allreduce`.

[191] Titolo, L., Feliú, M. A., Moscato, M., and Muñoz, C. A. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *Verification, Model Checking, and Abstract Interpretation* (Los Angeles, CA, USA, 2018), I. Dillig and J. Palsberg, Eds., VMCAI (LNCS 10747), Springer International Publishing, pp. 516–537.

[192] Torlak, E. Symbolic execution. Lecture Slides at the University of Washington, 2016. Available at `https://courses.cs.washington.edu/courses/cse403/16au/lectures/L16.pdf`.

[193] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[194] United States Department of Defense. Military standard: Sixteen-bit computer instruction set architecture. Tech. Rep. MIL-STD-1750A, United States Department of Defense, 1980.

[195] Vanhoef, M., and Piessens, F. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, TX, USA, Oct. 2017), CCS '17, ACM, pp. 1313–1328.

[196] Villa, O., Gurumoorthi, V., and Krishnamoorthy, S. Effects of floating-point nonassociativity on numerical computations on massively multithreaded systems. In *CUG 2009 Proceedings* (2009), Cray User Group, pp. 1–11.

[197] von Plato, J. The development of proof theory. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., winter 2018 ed. Metaphysics Research Lab, Stanford University, 2018.

[198] Wang, Q., Zhang, X., Zhang, Y., and Yi, Q. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 25:1–25:12.

[199] Watt, C. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA, Jan. 2018), CPP, ACM, pp. 53–65.

[200] Whitehead, N., and Fit-Florea, A. Precision and performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Tech. rep., NVIDIA Corporation, Santa Clara, CA, United States, Oct. 2018. Available at `https://docs.nvidia.com/pdf/Floating_Point_on_NVIDIA_GPU.pdf`.

[201] Wiles, A. Modular elliptic curves and fermat's last theorem. *Annals of Mathematics 141*, 3 (1995), 443–551.

[202] Wilkinson, J. H. *Rounding Errors in Algebraic Processes*. Dover Publications, Inc., Mineola, NY, USA, 1963.

[203] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. Formal methods: Practice and experience. *ACM Computing Surveys 41*, 4 (Oct. 2009), 19:1–19:36.

[204] Working Group for Floating-Point Arithmetic. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[205] Zhao, J., Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA, Jan. 2012), POPL '12, ACM, pp. 427–440.

[206] Zheng, M., Rogers, M. S., Luo, Z., Dwyer, M. B., and Siegel, S. F. CIVL: Formal verification of parallel programs. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2015), ASE '15, IEEE Computer Society, pp. 830–835.