2018 IEEE International Parallel and Distributed Processing Symposium

Do Developers Understand IEEE Floating Point?

Peter Dinda Conor Hetland
Northwestern University

***Abstract*—Floating point arithmetic, as specified in the IEEE standard, is used extensively in programs for science and engineering. This use is expanding rapidly into other domains, for example with the growing application of machine learning everywhere. While floating point arithmetic often appears to be arithmetic using real numbers, or at least numbers in scientific notation, it actually has a wide range of gotchas. Compiler and hardware implementations of floating point inject additional surprises. This complexity is only increasing as different levels of precision are becoming more common and there are even proposals to automatically reduce program precision (reducing power/energy and increasing performance) when results are deemed "good enough." Are software developers who depend on floating point aware of these issues? Do they understand how floating point can bite them? To find out, we conducted an anonymous study of different groups from academia, national labs, and industry. The participants in our sample did only slightly better than chance in correctly identifying key unusual behaviors of the floating point standard, and poorly understood which compiler and architectural optimizations were non-standard. These surprising results and others strongly suggest caution in the face of the expanding complexity and use of floating point arithmetic.**

***Keywords*-floating point arithmetic, software development, user studies, correctness, IEEE 754**

I. Introduction

Complete reliance on floating point arithmetic, as defined in the IEEE 754[6], and 754-2008 [7] standards, is a common denominator of most programs in science and engineering, ranging in scale from one-off scripts on laptops to long-running simulations on capability supercomputers. As more and more fields embrace a computational approach, leveraging HPC via simulation or data science, this reliance is expanding rapidly. As large scale machine learning is being applied to an exploding range of problems, this reliance is expanding well outside of science and engineering. The number of developers tasked with writing the needed programs is similarly expanding quickly.

The ubiquity and exploding use of floating point has also focused architecture, systems, and programming language/compiler researchers on optimizing floating point for performance and energy efficiency. Examples include offering lower precision but faster floating point numbers in the hardware (e.g. half-floats), considering non-IEEE compliant hardware [4], automatically reducing programmer-specified precision to the minimum possible to stay within error bounds [13], and approximate computing [9] where performance/energy and output quality can be traded off. More immediately, any programmer using a modern compiler is faced with dozens of flags that control floating point optimizations which could affect results. Optimizing a program across the space of flags has itself become a subject of research [5].

The floating point arithmetic experienced by a software developer via a particular hardware implementation, language, and compiler, is swelling in complexity at the very same time that the demand for such developers is also growing. This may be setting the stage for increasing problems with numeric correctness in an increasing range of programs. Numeric issues can produce major effects. Recall that Lorenz's insight, a cornerstone of chaos theory, was triggered by a seemingly innocuous rounding error [10]. Arguably, modern applications, certainly those that model systems with chaotic dynamics, could see small errors in developer understanding of floating point become amplified into bad overall results.

Do software developers understand the core properties of floating point arithmetic? Do they grasp which optimizations might result in non-compliance with the IEEE standard? How suspicious are they of a program's results in light of the various exceptions in the standard? What factors in a developer's background lead to better understanding and appropriate suspicion of results? We attempt to address these questions through a study of software developers. Our contributions are as follows:

- An anonymous survey taken by 199 software developers, largely from the sciences and engineering, plus an additional survey on suspicion taken by 52 non-overlapping students with controlled training.
- An analysis of the data from the surveys: Developers do little better than chance when quizzed about core properties of floating point, yet are confident. Developers do express a lack of confidence in dealing with optimizations, however. Both developers and students are less wary of certain exceptions than they ought to be. While several factors enhance developer performance, we did not find any particularly strong factor.
- A set of broader observations and potential actions that the community could take to improve the situation.

Related work: As far as we are aware, this is the first study of software developer understanding of floating point, hence there is little directly related work to point to. More incidentally related research is cited and discussed throughout the paper.

This project is made possible by support from the United States National Science Foundation through grant CCF-1533560 and from Sandia National Laboratories through the Hobbes Project, which is funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the United States Department of Energy's Office of Science.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00068

II. Survey design

Several requirements informed the design of our survey:

- Anonymity: Making individuals identifiable would reduce their likelihood of participation, particularly if they felt, rightly or wrongly, that their understanding of floating point was less than ideal.
- Low time commitment: Reducing the time to complete the survey would increase participation.
- Approximation of practice: We want to test for the application of floating point concepts within software development, *not* for knowledge of terminology.
- Avoidance of prompting and anchoring: A question should not lead the participant through use of terminology or other forms, nor should consideration of a question feed to a subsequent question.
- Factor identification: We want to be able to condition survey results on factors that could conceivably influence understanding of floating point, and suggest ways to increase such understanding among future users.

These requirements imply tradeoffs. For example, more factors decrease anonymity, and a lower time commitment reduces the ability to approximate practice.

We developed a survey design that would take less than 30 minutes to complete. We then tested the initial version on a group of 20 non-anonymous participants and gathering their feedback. This prompted a revision of the survey design to clarify several questions, and the introduction of several additional questions to tease apart situations where one question was inadvertently testing more than one concept.

Our ultimate survey consists of four components, as described below. Each bullet point in this section corresponds to one question in the survey. Note that we label each question here for convenience in our discussion, but no labels appear in the actual survey. The precise survey we used and other study documents can be found at http://presciencelab.org/float. The survey was online and implemented using Google Forms. It is currently a publicly announced open survey. However, for the results given in this paper, it was not public and we used a restricted recruitment model as described in Section III to garner participants who are most likely to use floating point in consequential positions in scientific research and engineering.

A. Background

The first component of the survey captures self-identified information about the relevant background of the participant. We first ask about the participant's general background and training in floating point.

- **Position:** The participant's current position (e.g., faculty, research scientist, research staff, software engineer, manager, post doc, Ph.D. student, etc.).
- **Area:** The participant's area of formal training (e.g., CS, CE, EE, other engineering, other physical science, mathematics, etc.).
- **Formal Training:** The amount of formal training *about floating point* that the participant reports receiving.
- **Informal Training:** The kinds of informal training *about floating point* the participant reports having used.

We also assess the participant's level of software development experience, and the extent of their interaction with floating point within that experience.

- **Software Development Role:** How does the participant view the software development they perform? Is their main role a software engineer, a manager of software engineers, or do they develop software or manage software development in support of their main role?
- **Floating Point Language Experience:** The participant reports the set of languages within which they have used IEEE floating point. A wide range of options are provided, and the participant can add their own.
- **Arbitrary Precision Language Experience:** The participant reports the set of languages that they have used that support non-floating point arbitrary precision numbers and arithmetic. A large number of options are provided, and the participant can add their own.
- **Contributed Codebase Size:** The participant reports the number of lines of code of the largest codebase they built or of their largest contribution to a shared codebase that they have made. This is captured by order of magnitude.
- **Contributed Codebase Floating Point Extent:** The participant indicates the extent to which floating point was involved in this codebase and their work within it. For involvement: not at all, incidental, and intrinsic. For intrinsic involvement of floating point, we also capture whether numeric correctness was the participant's focus, their team's focus, or the focus of another team.
- **Involved Codebase Size:** The participant reports the number of lines of code of the largest codebase they have been involved with in any capacity. This is captured by order of magnitude.
- **Involved Codebase Floating Point Extent:** This captures the extent to which floating point was involved in this codebase and their work within it.

B. Core quiz

The next component of the survey is a quiz in which each question generally shows a snippet of code in C syntax (C++, C#, and Java have identical syntax for each snippet) and makes an assertion. The participant is asked to choose whether the assertion is true, false, or that they do not know. The idea is to invoke the experience of writing code that uses floating point arithmetic to approximate real number arithmetic, and, in this context, reaching points where the participant has to consider floating point arithmetic behavior's divergences from the behavior of real number arithmetic. Note again that the labels we use here for convenience in discussion and do not appear in the survey.

- **Commutativity:** Is a simple statement involving the commutativity over addition true for floating point? Generally, floating point arithmetic follows the same commutativity laws as real number arithmetic. The question indicates that we are speaking of non-NaNs, although the term "NaN" is not used in order to avoid prompting or anchoring the participant.
- **Associativity:** Is a simple statement involving associativity over addition true for floating point? Generally, floating point arithmetic does *not* follow the associativity laws of real number arithmetic. We indicate we are speaking of non-NaNs without using that term. Misjudging associativity is a common source of problems.
- **Distributivity:** Is a simple statement involving distributivity of multiplication over addition always true for non-NaN floating point numbers? Such distributivity is always true of real number arithmetic, but *not* of floating point arithmetic, and in general the distributivity laws for real number arithmetic do not apply in floating point arithmetic. Misunderstanding distributivity is a common source of problems.
- **Ordering:** Is a simple statement involving ordering of operations for non-NaN number, specifically `((a + b) - a) == b`, always true for floating point arithmetic? This is *not* always true for floating point arithmetic, due to the existence of infinities and rounding, among other reasons, and misunderstanding ordering effects is a common source of problems.
- **Identity:** Is it always true that `a==a` if `a` is a floating point value (NaNs are now included, although the explicit term is still avoided)? Surprisingly, this is *not* always true in floating point arithmetic.
- **Negative Zero:** Given two floating point values that are both zero, is it possible for them not to be equal? The existence of "negative zero" in the standard implies this might be possible. However, it is not.
- **Square:** Is the square of a non-NaN floating point number always ≥ 0? This is true for both floating point arithmetic and real-number arithmetic, but not for integer computer arithmetic. This tests whether a participant confuses integer and floating point arithmetic, which are often learned together.
- **Overflow:** Both floating point arithmetic and computer integer arithmetic include the concept of overflow, but they behave completely differently. Overflow in computer integer arithmetic indicates wrap-around, while overflow in floating point indicates saturation at an infinity. This question determines whether the participant has confused these two behaviors.
- **Divide By Zero:** This question asks whether in floating point, $1.0/0.0$ is a non-NaN value. It is (infinity). The purpose here is to determine if the participant may believe that such an operation in their code might become immediately obvious by generating a NaN. Unlike a NaN, an infinity might ultimately propagate to the program output as an ordinary numeric value, disguising such an error.
- **Zero Divide By Zero:** This question asks whether in floating point $0.0/0.0$ is a non-NaN value, which it is not. NaN generation is desirable here, since it will propagate to the output as a NaN and thus make the user suspicious.
- **Saturation Plus:** This questions asks whether in floating point, it is possible for `(a + 1.0) == a` to be true. This is possible, since floating point arithmetic is saturating arithmetic, and thus `a` could be an infinity. Our goal here is to see if the user understands the existence of saturation. Another possible reason is rounding, if `a` has a very large magnitude. Misunderstanding saturation is a common source of errors.
- **Saturation Minus:** This is another variant of Saturation Plus and serves a similar goal, namely whether it is possible to "back off" from an infinity.
- **Denormal Precision:** This question asserts that floating point numbers very near zero have less precision than those further away. These numbers, the denormalized numbers, do have fewer bits of precision as they get closer to zero. The purpose of the question is to test if the participant is aware of denormalized numbers and their gradual underflow behavior (without specifically using such terms). Applications that rely on very small magnitude numbers must take such precision loss into account. Furthermore, some hardware has the ability to disable denormalized numbers, which can produce very different results in such applications.
- **Operation Precision:** This question asserts that floating point arithmetic operations can have results that have lower precision than the operands. This is true due to rounding. Our purpose is to test if the participant is aware of how rounding can produce different results than would be expected in real number arithmetic.
- **Exception Signal:** This question asserts that any floating point operation that delivers an exceptional result (NaN, infinities, etc) will inform the participant's application by default. This in fact does *not* happen in floating point arithmetic, and an additional source of confusion is that it is *partially* true for computer *integer* arithmetic. We want to see if the participant thinks that a signal-free program execution means no exceptional floating point value was generated. A participant who believes this has a very false sense of security.

C. Optimization quiz

It is very common for developers to seek to optimize their code for performance through compiler optimizations and hardware features. Some of these go beyond the IEEE standard. As a consequence, correct code can become incorrect under some optimizations and/or on some hardware. In this component of the survey, we try to determine if the

participant understands when they are choosing non-standard optimizations or hardware.

- **MADD:** Is the common fused multiply-add instruction, which the question describes, part of the standard? MADD is included in the newer IEEE standard, but not the original. MADD can compute a different result than separate multiplication and addition.
- **Flush to Zero:** Intel processors have control bits (FTZ and DAZ) that eliminate denormalized numbers and their "gradual underflow" behavior in favor of speed. On some hardware, the bits are on by default, leading to surprises particularly if very small (in magnitude) numbers are important to the computation. This is not part of the standard.
- **Standard-compliant Level:** Typical compilers have options for different levels of optimization (i.e., `-O3`). Which level is generally considered to the be the highest possible that still preserves standard-compliant floating point behavior? This is typically `-O2`, with `-O3` also allowing MADD.
- **Fast-math:** Compilers typically have a tempting `--ffast-math` option that enables certain optimizations. This question asks whether this can result in non-standard-compliant behavior. It can.

D. Suspicion quiz

Floating point hardware internally tracks exceptions for every operation via sticky condition codes. By default, these exceptions do not propagate to the application. In this component of the survey we pose a hypothetical situation in which we wrap a scientific simulation with code that determines if any of the possible exceptions occurred one or more times during the execution of the simulation. Then, for each of these possible exceptions, we ask the user to indicate, on a 5 point Likert scale, how suspicious they would be of the simulation results if the exception occurred. The possible exceptions are:

- **Overflow:** The result of an operation was an infinity. Arguably, this is usually a sign of trouble in real code.
- **Underflow:** The result of an operation was a zero. This is probably not a sign of trouble in real code.
- **Precision:** The result of an operation required rounding and thus a loss of precision. In practice, rounding is very common, and not a problem if the numeric behavior of the algorithm has been designed correctly.
- **Invalid:** The result of an operation was a NaN. This is almost invariably a sign of serious trouble in real code.
- **Denorm:** The result of an operation was a denormalized number. Similar to rounding, this is common, and not a problem given appropriate numeric algorithm design. However, if the developer does not expect to see very tiny, but non-zero results at any point of the computation, it could be a sign of trouble.

III. Participant recruitment and background

In recruiting participants we sought individuals at the Ph.D. student level and above who were actively involved in software development or the management of software development for science and engineering fields at universities, national labs, and industry. These individuals' understanding of floating point is likely to be critical to the correctness of scientific results based on the computational mode of investigation, and on the quality of engineered artifacts. We are not only interested in scientists/engineers who program, but also in the software engineers who program in support of science/engineering. In both cases, we would expect them to have a higher level of understanding of floating point than the general population of programmers because it is so intrinsic to their work.

Recall that an important goal of the survey design (Section II) is anonymity. Combining anonymity with the need to target a restricted population makes recruitment challenging. To recruit participants, we used a standardized email to announce the survey, ask the recipient take it, and request that the recipient forward it internally to appropriate groups. We specifically noted in the email that we were seeking Ph.D. students and above, in all fields.

Our set of initial "seed" recipients was chosen based on our existing connections and were high-level individuals that we felt could be relied on to forward the request reliably within their organizations. Essentially, we wanted to be confident that no initial recipient would simply spam a broad group that would include individuals outside of the restricted population we wanted to study, but rather would forward the request with care. Our initial recipients included:

- Chairs of all engineering and physical science departments at Northwestern University. Chairs of social science departments that use computation were also included. This was a total of 21 departments.
- Associate director for research in Northwestern's IT department, who oversees our shared HPC resources.
- Faculty, Ph.D. students, and postdocs within Northwestern's EECS department.
- Members of Northwestern's Center for Interdisciplinary Exploration and Research in Astrophysics (CIERA).
- Managerial contacts at the following national labs: Oak Ridge, Sandia, Argonne, and Los Alamos. These contacts are all involved in computational science and engineering, either directly or in a supporting role).
- Faculty contacts in CS/CE departments at the following universities: University of Chicago, Indiana University, IIT, University of Pittsburgh, University of Florida, University of New Mexico, Ohio State, University of Southern California / ISI, University of Queensland, Eidgenoessische Technische Hochschule Zuerich, Carnegie Mellon, Technische Universiteit Delft, University of Michigan, University of Minnesota, Georgia Tech, Virginia Tech, Rutgers University, Purdue Uni-

Position	n	%
Ph.D. student	73	36.7
Faculty	49	24.6
Software engineer	23	11.6
Research staff	17	8.5
Research scientist	11	5.6
M.S. student	8	4.0
Undergraduate	7	3.5
Postdoc	4	2.0
Manager	3	1.5
Other	5	2.5

Figure 1: Positions of participants.

Area	n	%
Computer Science	80	40.2
Other Physical Science Field	38	19.1
Other Engineering Field	26	13.1
Computer Engineering	19	9.5
Mathematics	10	5.0
Electrical Engineering	9	4.5
Economics	2	1.1
Other Non-Physical Science Field	2	1.1
CS&Math	2	1.1
CS&CE	2	1.1
Political Science and Statistics	1	0.5
Social Sciences	1	0.5
Robotics	1	0.5
Econometrics	1	0.5
Biomedical Engineering	1	0.5
MMSS	1	0.5
Statistics	1	0.5
Mechanical Engineering	1	0.5
Unreported	1	0.5

Figure 2: Areas of participants.

versity, Notre Dame, and Texas A&M, These contacts are all involved in HPC or closely-related research.

It is important to understand that given the anonymity of our survey, we cannot determine who specifically at these institutions participated in the survey or to whom our seed recipients forwarded our request to within their institutions or outside of them.

We attracted 199 participants through this process. To categorize them, despite anonymity, we rely on their self-reported background (Section II-A). Figures 1 through 11 describe our participants' backgrounds in detail.

Formal Training *in Floating Point*	n	%
One or more lectures in course	62	31.2
None	52	26.1
One or more weeks within a course	49	24.6
One or more courses	35	17.6
Not reported	1	0.5

Figure 3: Formal Training *in floating point* of participants.

Informal Training *in Floating Point*	n	%
Googled when necessary	138	69.4
Read about it	136	68.3
Discussed with coworkers/etc	89	44.7
Trained by adviser/mentor	38	19.1
Watched video	22	11.1

Figure 4: Informal Training *in floating point* of participants (Top 5 shown).

Software Development Role	n	%
I develop software to support my main role	119	59.8
My main role is as a software engineer	50	25.1
I manage others who develop software to support my main role	19	9.5
My main role is to manage software engineers	6	3.0
Not Reported	5	2.5

Figure 5: Software Development Roles of participants.

Floating Point Languages Experience	n	%
Python	142	71.4
C	139	69.9
C++	136	68.3
Matlab	105	52.8
Java	100	50.3
Fortran	65	32.7
R	48	24.1
C#	26	13.1
Perl	25	12.6
Scheme/Racket	17	8.5
Haskell	12	6.0
ML	9	4.5
JavaScript	6	3.0

Figure 6: Floating Point Language Experience of participants. 55 languages were reported. These 13 had $n \geq 5$.

About 1/3 of our participants are Ph.D. students, 1/4 are faculty, and 1/4 are software engineers, research scientists or staff. About 1/2 were formally trained in computer science or engineering, and 1/2 were formally trained in a science or engineering field, including math (5%). Almost 2/3 develop software to support their main role, while about 1/4 see software engineer as being their primary role.

More than 3/4 report some formal training about floating

Arb. Precision Language Experience	n	%
Mathematica	71	35.7
Maple	29	14.6
Other language	20	10.0
MPFR/GNU MultiPrecision Library	19	9.6
Scheme/Racket/LISP with BigNums	13	6.5
Other library	13	6.5
Matlab MultiPrecision Toolbox	10	5.0
Haskell with arb. prec. and rationals	8	4.0
Macsyma	5	2.5

Figure 7: Arbitrary Precision Language Experience of participants. 38 languages/libraries were reported. These 9 had $n \geq 5$.

Contributed Codebase Size	n	%
1,001 to 10,000 lines of code	79	39.7
10,001 to 100,000 lines of code	65	32.7
100 to 1,000 lines of code	27	13.6
100,001 to 1,000,000 lines of code	17	8.5
>1,000,000 lines of code	9	4.5
<100 lines of code	1	0.5
Not Reported	1	0.5

Figure 8: Contributed Codebase Sizes of participants.

Contributed Codebase Floating Point Extent	n	%
FP incidental	77	38.7
FP intrinsic	63	31.7
FP intrinsic, I did numerical correctness	29	14.6
FP intrinsic, other team did numerical correctness	10	5.0
FP intrinsic, my team did numeric correctness	10	5.0
No FP involved	9	4.5
No Report	1	0.5

Figure 9: Contributed Codebase Floating Point Extent of participants (within the codebase they built (Figure 8.)

Involved Codebase Sizes	n	%
10,001 to 100,000 lines of code	61	30.7
1,001 to 10,000 lines of code	53	26.6
>1,000,000 lines of code	36	18.1
100,001 to 1,000,000 lines of code	36	18.1
100 to 1,000 lines of code	8	4.0
<100 lines of code	2	1.0
No Report	3	1.5

Figure 10: Involved Codebase Sizes of participants.

point, most commonly one or more lectures in a course. Almost all report informal training about floating point, with Googling and reading being the most common. Depressingly, less than 20% report training from their adviser or mentor. The participants have experience with floating point in 55 different languages, with Python, C, C++, Matlab, Java, and Fortran each being reported by 1/3 or more. Over 2/3 have experience with arbitrary precision languages/libraries, with Mathematica being by far the most common (over 1/3).

Almost 1/2 of our participants have personally written a codebase or made a codebase contribution of at least 10,000 lines of code, and floating point was intrinsic to almost 2/3 of those codebases. Over 2/3 have been involved with a codebase of at least 10,000 lines of code, and floating point was intrinsic to over half of those codebases. Less than 8% reported codebases in which floating point was not involved.

Of course, our analysis results depend on the nature of the sample. We believe that the combination of our recruitment process and the resulting background of the participants illustrated here suggest that our sample is a good representative of software developers who write code for, and in support of, science and engineering applications.

Additional participant group for suspicion quiz: We also administered the suspicion quiz (Section II-D) to a group of 52 undergraduates at Northwestern. These students were taking EECS 213, Introduction to Computer Systems, which includes book [3], lecture (one week / 160 minutes), lab (∼1/8 of lab content in quarter), and homework (∼1/8 of homework content in quarter) material on floating point. The course material did *not* include the floating point condition codes (other than the x64's oddball use of the parity bit to indicate a NaN result). Note that this training is similar in lecture quantity (one or more weeks) to about 1/4 of our previous study population (Figure 3). The suspicion quiz was given as a midterm exam problem. There were no wrong answers, although the students did not know this when taking the exam. This data gives us a comparison group for the suspicion quiz for whom we know precisely the content of the formal training given.

Involved Codebase Floating Point Extent	n	%
FP incidental	71	35.7
FP intrinsic	55	27.6
FP intrinsic, I did numerical correctness	23	11.6
FP intrinsic, other team did numerical correctness	17	8.5
No FP involved	15	7.5
FP intrinsic, my team did numeric correctness	13	6.5
No Report	5	2.5

Figure 11: Involved Codebase Floating Point Extent of participants within the largest codebase they were involved with (Figure 10).

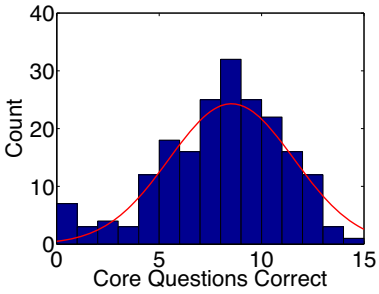


Figure 13: Histogram of core quiz scores. There are 15 questions. Chance would put the mean at 7.5.

IV. Analysis results

We analyzed the dataset in a wide variety of ways with the following questions in mind:

- Do developers understand floating point arithmetic in terms of how it differs from real arithmetic and computer integer arithmetic?
- Do developers understand how optimizations at the hardware and compiler level may affect the behavior of floating point arithmetic within or beyond the standard?
- What are the common misunderstandings?
- What factors have an effect on understanding?
- What might make developers suspicious of a result?

We now summarize the main results of our analysis.

A. General understanding

The most important results of our analysis are in Figure 12. Here, we show what the average (i.e. expected) score was on the core and optimization quizzes. Our participants generally feel they *can* answer the core quiz questions, but then perform at near chance levels on those questions. The score for the core quiz was 8.5/15, which is only slightly better than would be expected by chance (7.5/15). Figure 13 shows a histogram of scores on the core quiz. There is a subtlety here in that "Don't Know" was a possible response to a question. The incidence of this, however, was $< 15\%$ for the core quiz. In contrast, in the optimization quiz, our

Core Quiz				
# Correct	# Incorrect	# Don't Know	# No Answer	**# Chance**
8.5	4.0	2.3	0.2	**7.5**
Optimization Quiz				
# Correct	# Incorrect	**# Don't Know**	# No Answer	# Chance
0.6	0.2	**2.2**	0.1	1.5

Figure 12: Average (expected) performance of participants on the core and optimization quizzes. For the core quiz, most participants were comfortable giving an answer for most questions, yet the expected number of correctly answered questions (8.5) is only slightly higher than would be expected by chance (7.5). For the optimization quiz, most participants answered "Don't Know" for most questions. Standard-compliant Level is not included as it is not a T/F question.

participants generally recognized their ignorance, answering "Don't Know" over 2/3 of the time.

The core quiz behavior is alarming, while the optimization quiz behavior is reassuring. Our participants seem to be appropriately wary about compiler and hardware optimizations (perhaps because that level of detail may not have been encountered before in their limited training (Figures 3 and 4), but overconfident in basic floating point behavior (which is encountered in these forms of training).

Figure 14 is a question-by-question breakdown of the core quiz. As highlighted in the table, 6/15 questions are answered at chance levels, while 2/15 are answered incorrectly by most participants. Notice also that while participants do better than chance on Associativity (69.3% correct), Overflow (60.8%), and Exception Signal (69.3%), these are not exactly stellar numbers for such important concepts. 30% of our participants may think that an exceptional floating point value (NaN, etc) will result in a signal, which it will not. Considering Overflow and the Saturation questions, as many as half of our participants may not fully appreciate that floating point arithmetic is saturating arithmetic instead of modular arithmetic.

Figure 15 is a question-by-question breakdown of the optimization quiz. The news here is that over 2/3 of participants reported that they did not know whether noted optimizations resulted in non-standard behavior. <10% (<20% with a more generous definition) knew at which optimization level the compiler could produce non-standard compliant code. Less than 1/3 knew that `-ffast-math`, the "least conforming but fastest math mode" to quote the gcc manual, could produce non-standard behavior.

B. Factor analysis for core quiz

We considered each of our background factors (Section II-A) as a predictor of performance on the core and optimization quizzes. We have enough data to meaningfully consider each factor in isolation, which we did. In the following, we report on the factors that seem to have the largest significance only.

Although several factors are somewhat predictive, no factor has an outsize impact on performance in the core quiz. In the best case, the average performance rises from 8.5/15 to 11/15, and the variation across the values of the factor is 4/15.

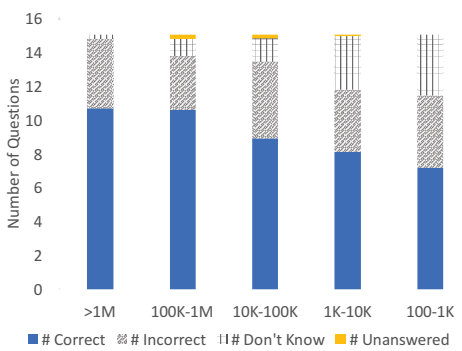


Figure 16: Effect of Contributed Codebase Size on core quiz scores.

Perhaps not surprisingly, the most predictive factor is simply Contributed Codebase Size, the effect of which is shown in Figure 16. The effect of Involved Codebase Size is similar. The larger the codebase that the participant has experienced or built, the better their understanding of floating point. However, this is no panacea. Even those who have built million line codebases are still getting an average of 4 out of 15 questions wrong.

One might expect that given the importance of Contributed Codebase Size (or Involved Codebase Size), that Contributed Codebase Floating Point Extent (and Involved Codebase Floating Point Extent), which measure the degree of the participant's interaction with floating point within these codebases, would be high. While there is an effect—if the participant or their team focused on numeric correctness within a codebase, they are more likely to have a higher score—it is a small effect. There is a gain of only about 2/15 compared to those who reported a codebase where floating point was not intrinsic or where they were not involved.

Area follows Codebase Size closely as a predictive factor, and is illustrated in Figure 17. It may not be surprising that participants from areas closest to the construction of floating point (EE, CS, CE) do better, but note that this at best raises average performance from 8.5/15 to 11/15 and the variation across the values is 3.5/15. What is particularly disturbing is that "Other Physical Science Field" (PhysSci) and "Other Engineering Field" (Eng) are performing at the level of chance. Yet developers from these areas are likely to be the most extensive users of floating point!

The effect of the participant's Software Development Role

Question	% Correct	% Incorrect	% Don't Know	% Unanswered
Commutativity	53.3	27.6	18.6	0.5
Associativity	69.3	14.1	15.6	1.0
Distributivity	81.9	6.0	10.6	1.5
Ordering	80.4	6.0	12.6	1.0
Identity	16.6	76.9	5.5	1.0
Negative Zero	58.8	28.1	11.6	1.5
Square	47.2	35.2	16.6	1.0
Overflow	60.8	24.1	11.1	4.0
Divide by Zero	11.6	76.4	11.1	1.0
Zero Divide By Zero	70.4	9.0	19.6	1.0
Saturation Plus	54.8	26.1	17.6	1.5
Saturation Minus	53.3	25.6	19.6	1.5
Denormal Precision	52.3	24.6	22.1	1.0
Operation Precision	73.4	9.0	16.6	1.0
Exception Signal	69.3	10.1	19.6	1.0

Figure 14: Core quiz questions. Boldfaced questions were answered correctly at the level of chance. Italicized questions were answered incorrectly or reported as unknown more often than answered correctly.

Question	% Correct	% Incorrect	% Don't Know	% Unanswered
MADD	15.6	10.0	72.4	2.0
Flush to Zero	13.6	7.5	76.9	2.0
Standard-compliant Level	8.5	20.7	68.8	2.0
Fast-math	29.1	3.0	65.8	2.0

Figure 15: Optimization quiz questions. All questions were reported as unknown by more than half the participants.

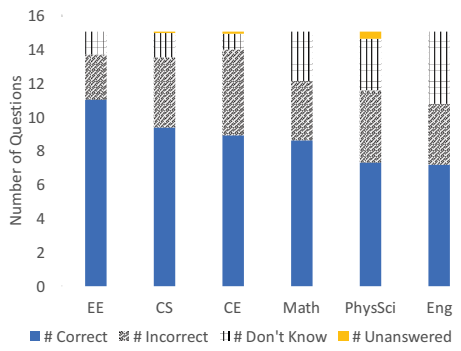


Figure 17: Effect of Area on core quiz scores.

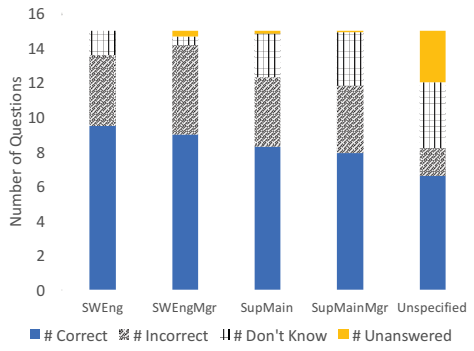


Figure 18: Effect of Software Development Role on core quiz scores.

follows in importance, and is illustrated in Figure 18. Those who view their main role as software engineering do slightly better than those who see their software engineering as done in support of their main role.

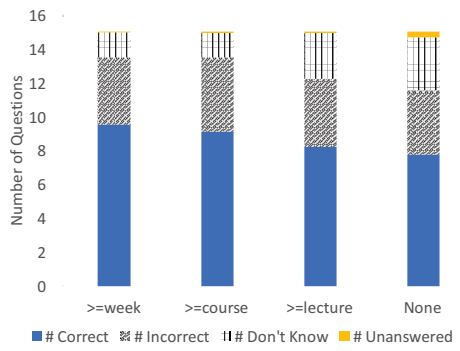


Figure 19: Effect of Formal Training (in floating point) on core quiz scores.

One might hope that Formal Training (in floating point) would have a considerable effect. While it does have an effect (as illustrated in Figure 19, it is not a large one. The maximum gain over the baseline is only about 1/15, and the variation is about 2/15.

The factors we have not discussed thus far, Position, Informal Training (in Floating Point), Floating Point Language Experience, and Arbitrary Precision Language Experience, have minimal or ambiguous effects on core quiz scores. Note that the participant supplies lists for some of these factors. What seems to be true about them is that very short lists predict bad scores. That is provided the participant has reported *some* kind of Informal Training, etc, it does not seem to matter what it is. A possible explanation is that these are driven by the two Codebase Size factors. If a participant wrote or was involved with a codebase of any significant size, they probably experienced a range of languages and

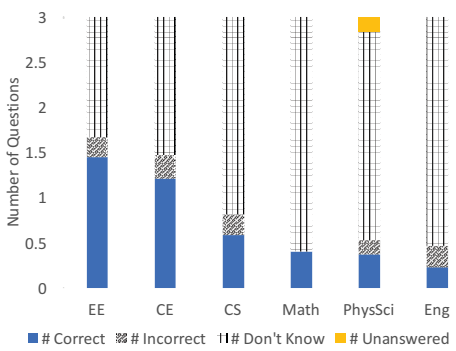


Figure 20: Effect of Area on optimization quiz scores.

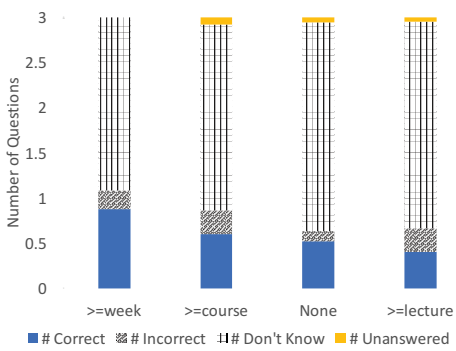


Figure 21: Effect of Software Development Role on optimization quiz scores.

had to participate in some kind of informal training.

C. Factor analysis for optimization quiz

The main story about the optimization quiz is the dominance of the response "Don't Know" regardless of how the data is sliced by the factors.

Only the factors Software Development Role (Figure 21) and Area (Figure 20) appear in our data to have an effect on optimization quiz scores. Even there, the effects cap quickly (0.7/3 above chance for Role and 0.5 above chance for Area), although the variation is considerable (1.4/3 for Role and 0.8/3 for Area). The effect of Informal Training (in floating point) is ambiguous in our data, although it could be interpreted as producing considerable variation, albeit the maximum effect we see is only slightly better than chance.

It is surprising that factors like the codebase sizes and floating point experience within codebases have no effect here. On the other hand, the limited impact of Formal Training (in floating point) could be explained by the fact that most of those who have received training in our sample have received a level similar to that of an introductory computer systems or machine organization course. Such an introduction will not touch on optimizations at all.

D. Suspicion analysis

There is of course no ground truth for the suspicion quiz component of our survey, and it really depends on the application. However, as we described earlier, some exceptional conditions are *generally* more suspicious than others—an arguably reasonable ranking is that generating a NaN (Invalid) is by far more suspicious than generating an infinity (Overflow), which is in turn much more suspicious than generating any of the other three conditions.

Figure 22(a) shows the distribution of reported suspicion for the five exceptional conditions within our 199 participant main group, while Figure 22(b) shows the corresponding distribution for our separate 52 participant student group. The groups behave quite similarly, although the student group is overall less suspicious about Underflow and Denorm, possibly because the topic is fresh in their minds given the course. The student group is also less suspicious of Overflow.

As we might hope, both groups do tend to be more suspicious of Invalid and Overflow than the other conditions. However, consider Invalid more carefully: About 1/3 of both groups reported a suspicion level less than the maximum for a computation that somewhere encountered a NaN!

V. Conclusions

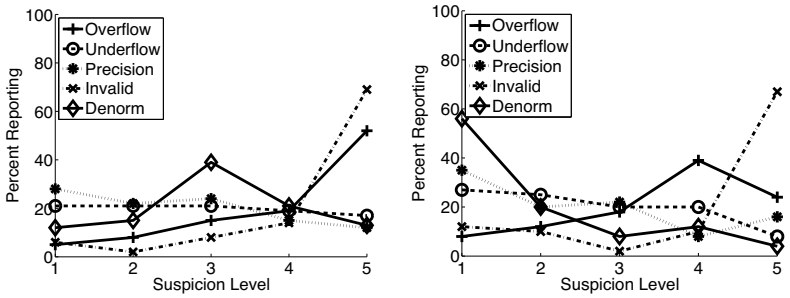
Stepping back from the data and analysis, we believe that some generalizations can be made, along with actions to address them.

Observation: Many developers do not understand core floating point behavior particularly well, yet believe they do. This suggests that some existing and future codebases may have hidden numeric correctness issues. This is probably more likely to be the case in smaller and newer projects where there is no specialist whose role is in part to mitigate these issues. As use of floating point rapidly expands outside of the traditional domains of science and engineering, the problem is likely becoming widespread.

Action: The HPC community should make an effort to make developers in general more suspicious about floating point behavior. The analogy might be how the programming languages and operating systems communities have raised awareness about C's undefined behavior and its interaction with modern compilers [12], [14].

Action: Although our study found that formal training in floating point has only a small effect on understanding, we believe the issue is not that training does not work per se, but rather that the community has just not found the right training approach yet. A rigorous process to develop effective training for a broad range of developers is an action that the HPC community, for example via SIGHPC, could undertake. We would then also need to convince the broader (and ever expanding) non-CS community of developers that such training is necessary.

Action: Static and dynamic analysis tools that can examine existing codebases and point developers to potentially suspicious code would likely have significant impact. Several such tools exist [1], [11], [8], but the tools would also need



(a) Main Group ($n = 199$) (b) Student Group ($n = 52$)

Figure 22: Distribution of suspicion for different exceptional conditions.

to have interfaces suitable for a non-CS community and have a low barrier to use. Perhaps commercial tools like Coverity [2] will expand their purview to include floating point. We ourselves have been developing a simple runtime monitoring tool to spy on unmodified binaries and track exceptional conditions using floating point condition codes, similar to the structure of the suspicion quiz.

Action: The boundary between floating point and arbitrary precision arithmetic is too thick. A system that would allow code written using floating point to be seamlessly compiled to use arbitrary precision would enable developers to easily sanity check the behavior of their code (and any optimizations they chose). A particularly paranoid developer could just opt for slow, arbitrary precision results.

Observation: Many developers recognize their lack of knowledge of how hardware and software optimizations affect floating point behavior. As the space of such optimizations expands, it could be that developers simply use them without understanding the consequences, or developers could simply avoid them out of fear of incorrect results, which would reduce their impact. There may be a parallel with the OS developer community, where optimizations that leverage C's undefined behavior are carefully avoided lest they break working kernel code or make it insecure.

Action: We need to assess to what extent developers wittingly or unwittingly *use* hardware and software optimizations without knowing their consequences. Are they as conservative about what they use as they are about what they think they know? If not, then the introduction of optimizations may be leaving a hidden trail of incorrect results behind it.

Action: Optimization implementations should take developer knowledge into account—ideally, a developer would not be able to use an optimization without demonstrating that they understand it. How can we create an effective interface for this that would not be gameable or too onerous to use?

REFERENCES

[1] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commuications of the ACM*, vol. 53, no. 2, pp. 66–75, February 2010.

[3] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Pearson, 2015.

[4] M. Fagan, J. Schlachter, K. Yoshii, S. Leyffer, K. Palem, M. Snir, S. M. Wild, and C. Enz, "Overcoming the power wall by exploiting inexactness and emerging cots architectural features: Trading precision for improving application quality," in *Proceedings of the 9th IEEE International System-on-Chip Conference (SOCC)*, September 2016.

[5] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2008.

[6] IEEE Floating Point Working Group, "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[7] ——, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[8] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, 2013.

[9] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, Mar. 2016.

[10] F. C. Moon, *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers*. John Wiley and Sons, Inc., 1992.

[11] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[12] J. Regehr, "Embedded in academia," https://blog.regehr.org/, a long running series of posts on undefined behavior has been widely read.

[13] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, 2013.

[14] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.