NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2021-07

December, 2021

Comparing the Understanding of IEEE Floating Point Between Scientific and Non-scientific Developers

Peter Dinda Alex Bernat

Abstract

IEEE floating point arithmetic and its implementations can be con- fusing, making developer understanding critical for correctness combined with performance. Our earlier work studied developer understanding of floating point through a survey instrument, targeting a closed, anonymous scientific developer population. It found evidence of a lack of understanding of IEEE floating point, with average performance of the group being slightly better than chance. In this work, we apply the same (open) survey instrument to an open, anonymous software developer population. We analyze the results of this survey using the same tools as our previous work, and compare the two populations. We find that the software developers have a better understanding of the gotchas in the IEEE floating point standard and a much better grasp of several implementation details. However, within the software developer group, demographic factors have limited power in explaining differences. [*]

Keywords

Floating Point Arithmetic, User Studies

[*] This project was supported by the United States National Science Foundation via grants CCF-1533560, CNS-1763743, and CCF-2028851.

Comparing the Understanding of IEEE Floating Point Between Scientific and Non-scientific Developers

Peter Dinda
Northwestern University

Alex Bernat
Northwestern University

ABSTRACT

IEEE floating point arithmetic and its implementations can be confusing, making developer understanding critical for correctness combined with performance. Our earlier work studied developer understanding of floating point through a survey instrument, targeting a closed, anonymous scientific developer population. It found evidence of a lack of understanding of IEEE floating point, with average performance of the group being slightly better than chance. In this work, we apply the same (open) survey instrument to an open, anonymous software developer population. We analyze the results of this survey using the same tools as our previous work, and compare the two populations. We find that the software developers have a better understanding of the gotchas in the IEEE floating point standard and a much better grasp of several implementation details. However, within the software developer group, demographic factors have limited power in explaining differences.

CCS CONCEPTS

• **Software and its engineering → Correctness**; **Software reliability**; **Operational analysis**; • **Mathematics of computing → Numerical analysis**; **Arbitrary-precision arithmetic**.

KEYWORDS

Floating point arithmetic, User studies

1 INTRODUCTION

Floating point arithmetic via the IEEE standard is ubiquitous in scientific computing and increasingly so in many other domains, such as machine learning. There is indeed increasing interest in alternative arithmetic systems such as posits, and unums, fixed point arithmetic, arbitrary precision arithmetic, as well as reduced precision and other limited versions of the IEEE standard. However, all current hardware implements the IEEE standard and all current compilers target these implementations and optimize for them—a number in a program that looks like real number is almost always an IEEE floating point number.

Unfortunately, IEEE floating point arithmetic is *not* real number arithmetic, and the similarity can be deceiving for developers. Common expectations of real number arithmetic simply do not hold true in IEEE floating point arithmetic and, consequently, programs can implement numeric algorithms incorrectly. Furthermore, even for correctly written source code, the plethora of optimization choices in a modern compiler can lead to differing results, and even usurp programmer intent. Hardware itself is increasingly loose in its adherence to the standard, leading to yet another possible source of error. Finally, while the standard provides hardware-level reporting mechanisms, the programmer must actually use these and understand their implications.

Recently, we published a first-of-its-kind study of developer understanding of IEEE floating point arithmetic [7]. Using a range of mechanisms, the study targeted 199 scientific developers from academia, national labs, and industry, while maintaining their anonymity. The results of the study, which we summarize in Section 2.3, were rather surprising. As a whole, these *scientific developers* did only slightly better than chance in correctly identifying key unusual behaviors of the IEEE floating point standard, poorly understood which compiler and architectural optimizations were non-standard, and were arguably less suspicious than they ought to have been when presented with hardware-level reporting about computations. For the optimizations, the participants were aware of their limitations, which could mean that optimizations are very conservatively applied.

In the present work, we applied precisely the same survey instrument to a separate, anonymous population of 352 individuals, drawn from an open call, who represent a broader set of developers, *software developers*. As Section 3 describes, we used a marketing methodology that commenced shortly after the prior work was presented in order to make our survey broadly available and visible. It is anonymous participants after the presentation date that are included in the software developers group that we analyze here.

We repeat the analysis of our previous work on the new software developers data set, present the results, and compare and contrast the results with those of the previous scientific developers data set. 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? How different are the answers to these questions from those of scientific developers? What are the implications? We address these questions through a study of software developers, and a comparison with the same study design applied to scientific developers. Our contributions are:

- An anonymous survey taken by 352 software developers, recruited on an open basis immediately after the results of an anonymous study of 199 scientific developers were presented.
- An analysis of the data from the survey that evaluates the software developer group with regard to the above questions, and compares their performance with that of the earlier scientific developer group.
- The surprising finding that the software developers group does considerably better than the scientific developer group

This project was supported by the United States National Science Foundation via grants CCF-1533560 and CNS-1763743, and CCF-2028851.

in terms of their understanding of the core aspects of the standard, and its optimizations.

- The surprising finding that the two groups do not differ in their (arguably insufficient) suspicion of potentially problematic floating point events.
- A finding that different factors are significant for the software developer group compared to the scientific developer group.

Related work. Correct and resilient computation using floating point arithmetic has, of course, been a long-standing topic of interest within the numerical methods, scientific computing, and software engineering communities. Finding floating point misbehavior in applications is a current hot topic, with a range of work that tries to find such issues, and then identify their root cause in the software, or ameliorate them more directly. Particular examples include Milroy et al [16], Herbgrind [18], and Flit [2, 19]. Milroy et al and Flit use variant compilation to induce output variation, and then localize the source of this variation to a root cause in the source. Herbgrind also attempts to find the root cause, although here the starting point can be instructions that have high rounding error. FPSpy [6] uses hardware to detect problematic floating point events in unmodified binaries. Programming language and compiler techniques for improving floating point behavior in applications is also a current hot topic (e.g. [1, 3, 5, 11, 12, 17]). A provocative recent proposal recommends creating an API for the real numbers [4].

Others have studied scientific software developers (e.g. [8–10, 13–15, 20, 21]), but as far as we are aware, this paper is only the second study of *software developer understanding of floating point*, hence there is no directly related work beyond our prior work whose survey instrument and analysis methodology we leverage [7].

2 SUMMARY OF PREVIOUS PAPER

Our previously published work [7] designed and tested a survey instrument that would allow anonymous participants to provide a snapshot of their background, and then complete a quiz with three components. We carefully targeted the survey instrument to an audience that was arguably rich in scientific developers, and analyzed the results. A more detailed description of the previous instrument, recruitment scheme, analyses, and results can be found in the previous paper, and is summarized here.

2.1 Survey instrument

The survey instrument, which we reuse in this work without changes, consists of three components.

Background. The background questions capture demographic information that may impact participants' knowledge of floating point. The survey first asks about background and training. It seeks to learn current employment position (eg., software engineer, Ph.D. student, or research scientists, etc.) and area of formal education (e.g., CS, CE, EE, Physical Science, etc.). The survey also asks about the amount of formal training the participant has received, specifically about floating point, as well as the types of informal training about floating point the participant has used.

The survey also probes participants' software development experience and use of floating point. To that end, it asks how the participant describes their role in the software development process (e.g., software engineer, developing software to support their main role, etc.), what languages the participant has experience using floating point in and what arbitrary precision languages the participant has used. Lastly, it asks the participants for the sizes of the largest codebases they have contributed to/been involved with, and the extent to which floating point was used in those codebases. For codebases in which floating point was intrinsic, the survey also captures whether correctness was the participant's focus, their team's focus, or another team's focus.

Core quiz. Next up are 15 questions in which the participant is shown a snippet of C code and asked whether an assertion about it is true, false, or if they do not know. The goal is to mimic the process of writing code that uses floating point arithmetic and force the participant to consider where floating point numbers do not act like real numbers.

The Commutativity, Associativity, and Distributivity[1] questions evaluate how the participant understands these aspects of floating point arithmetic, which behave differently than in real number arithmetic. Misconceptions here are common sources of problems.

The Ordering question tests the understanding of the consequences of rounding and saturating arithmetic on the order of operations suggested by equalities such as $((a + b) - a) == b$.

The Identity question asks if $a == a$ for all floating point numbers, including NaNs, which are not ordered with respect to the floating point numbers. This statement is surprisingly not true and can be a source of error. Similarly, the Negative Zero question asks if two values that are both zero are always equal. The existence of "negative zero" in the standard suggests they may not be, but two zeros are, in fact, always equal.

The Square question tests whether the participant is confusing integer and floating point arithmetic, as the square of a real or floating point number is always ≥ 0.0. This statement is not always true for integers because integer arithmetic is modular, not saturating. The related Overflow question determines whether participants confuse integer and floating point overflow.

The Divide by Zero question asks whether a floating point number divided by zero is a non-NaN value. It is actually an infinity, rather than an NaN. Does the participant expect such an operation to become obvious by generating a NaN? Unlike a NaN, an infinity may propagate to output as an ordinary value, disguising an error. The survey also asks about the behavior of Zero Divide by Zero, which does generate a NaN that can propagate to output and generate suspicion.

The Saturation Plus and Minus questions ask if $a + 1.0 == a$ or $a - 1.0 == a$ can ever be true. Because floating point arithmetic is saturating, such behavior is possible if a is an infinity. Rounding can also cause such behavior, if a has a very large magnitude.

The Denormal Precision question asks whether floating point numbers very close to zero have less precision than those further away. This question's goal is to find out if the participant is aware of gradual underflow behavior as applications relying on small magnitude numbers must account for precision loss. Also, some hardware can disable denormalized numbers for speed, which can produce unexpected results.

[1]The labels provided here are for convenience and do not appear on the survey.

The Operation Precision question asserts that the results of floating point operations can have less precision than the operands. This assertion is true because of rounding, which is an inherent part of most floating point operations.

The Exception Signal question asks whether any floating point operation that produces an exceptional result will inform the application by default. This is not true. Participants may be additionally confused because this statement partially holds for integer arithmetic. Participants who believe that a signal-free execution means no exceptional values were generated have a false sense of security.

Optimization quiz. Developers commonly seek to optimize their code with various compiler optimizations and hardware features. Correct code can become incorrect under certain optimizations and when using certain hardware features. This component of the survey tests participant knowledge of four optimizations (out of hundreds that exist), asking whether they result in behavior that is not in compliance with the IEEE standard.

The MADD question asks if the common `MADD` (fused multiply add) instruction is part of the standard. `MADD` is included in the newer version of the IEEE standard. `MADD` can return a different value than separate multiplication and addition instructions.

The Flush to Zero question asks about the control bits (FTZ and DAZ) on Intel processors that eliminate denormalized numbers and gradual underflow to increase speed. In some cases, these bits are active by default, which can lead to unexpected behavior with small magnitude numbers. FTZ and DAZ are not part of the standard.

The Standard-compliant Optimization Level question asks the highest possible typical compiler optimization level (ie., `-O3`) that remains compliant with the IEEE standard. The highest level is generally `-O2`, as `-O3` allows `MADD`s.

The Fast-math question asks whether the `--ffast-math` compiler flag's various optimizations can result in non-compliant behavior. They can.

Suspicion Quiz. Hardware tracks exceptions for every operation using sticky condition codes. By default, these exceptions do not appear in the application's results. The survey presents a scenario in which a program is wrapped to uncover exceptions (FPSpy [6] does this). It asks the user how suspicious they would be of the application's results, given they uncover a particular exception, on a five point scale.

The possible floating point exceptions include Invalid, which indicates an operation involved a NaN. This is almost always a sign of serious trouble. Overflow, which indicates the result of an operation was an infinity, also usually indicates trouble. Underflow, which indicates the result of an operation was a zero. Underflows are usually not indicative of problems. The Denorm exception indicates that gradual underflow occurred in some operation, or, equivalently, that denormalized numbers near zero were used. A Precision exception indicates an operation required rounding and therefore a loss of precision. Rounding is extremely common and is not usually indicative of a problem if the numeric behavior of the application has been designed correctly.

2.2 Demographics

The previous paper used a targeted closed recruitment process to try to maximize the number of scientific developers in its sample ($n = 199$). The demographics (which are also included for comparison in Section 3) reflect this goal, with a majority of participants describing themselves as Ph.D. students or faculty. About 1/3 of respondents were Ph.D. students and 1/4 faculty, with another 1/4 comprised of software engineers, research scientists and staff. About 1/2 of participants were trained in computer science or computer engineering. Because the previous survey targeted scientific developers rather than software developers it is unsurprising that nearly 2/3 develop software to support their main role, with only 1/4 developing software as their main role.

Over 3/4 of participants reported some formal training in floating point, with slightly under 1/3 of participants reporting that trainings took the form of one of more lectures in a course. Nearly all participants in the group reported some form of informal training in floating point, with googling being the most common method.

The participants had used floating point in 55 languages, with Python, C, C++, Matlab , and Java being reported by over 1/2 and Fortran being reported by about 1/3. Over 2/3 had experience with arbitrary precision libraries, with Mathematica being used by over 1/3 of participants and Maple being used by more than 1/8.

About 1/2 of participants contributed to codebases of over 10,000 lines and over 2/3 were involved in codebases containing 10,000 lines of code. Floating point was intrinsic to over 2/3 of contributed codebases and over 1/2 of involved codebases. About 7.5% of respondents reported codebases with no floating point use.

2.3 Main results

Participants generally believe that they know the answers to the Core quiz questions. However, their average score is 8.5 out of 15, which is *only slightly better than the chance would indicate*. Participants do better than chance on several key concepts (Associativity, Overflow, and Exception Signals). However, fewer than 70% of participants answered correctly on each of these questions. 6/15 core questions were answered at chance levels and 2/15 were answered incorrectly by most respondents. For the Optimization quiz, participants answered "Don't Know" 2/3 of the time, suggesting they are likely to be conservative in their use of optimizations. Notably, fewer than 10% of respondents correctly identified the optimization level at which a compiler could produce non-compliant code. In the Suspicion quiz, respondents were generally (and correctly) more suspicious of Invalid and Overflow, but 1/3 of participants reported a suspicion level less than the maximum for Invalid.

No background factor has an outsize impact on the Core quiz, even though several are somewhat predictive. The most predictive is Contributed Codebase Size, with the larger the code base, the better floating point understanding. However, participants with the largest codebases (>1M lines) still answered 4/15 questions incorrectly, on average. There is a slight gain of 2/15 questions when comparing scores from people who focused on numeric correctness in codebases and those who did not or where floating point was not intrinsic. Participants from areas closer to the floating point standards (CS, CE, EE), do slightly better (8.5/15 to 11/15). Participants in areas that may write scientific applications, like "Other

Physical Science Field" and "Other Engineering Field" perform at the level of chance, which is alarming. Formal training had only a small effect on performance.

The Software Development Role and Area factors have small effects on Optimization quiz results. The non-existent effect of Formal Training could be explained by participants' training being at the level of an introductory computer systems course, which does not touch on optimizations. Suspicion results were also broadly like those of a group of students ($n = 52$) in just such a course.

3 PARTICIPANTS

The previous paper, on the scientific developers group, was presented publicly on May 23, 2018. Immediately before the talk, we made the paper available. We also created a new open instance of the survey[2] at this point, and then proceeded to advertise it in the following manner:

- We advertised the new instance of the survey as part of our talk for the previous paper.
- We emailed all of the 1st degree contacts from the previous paper to make them aware of the new open survey. These contacts were not the participants in the original survey, but rather the individuals who selectively forwarded the authors' announcement to people who they felt were scientific developers. We now told them to forward to anyone.
- Our lab, project, and personal websites were updated to include pointers to the new open study.
- We created a post on Slashdot (http://slashdot.org), a venerable programming news aggregator, announcing the open survey.
- We facilitated a news article on our work in our university's print+web publication, which is distributed to about 255,000 people. A link to the open survey was included.
- A Hacker News (https://news.ycombinator.com/) discussion thread about the previous paper ensued. Hacker News is a widely read news aggregator for software developers and others. We made sure the URL for the open survey was highly visible in that discussion.
- A Reddit (https://reddit.com) thread also ensued, based on the Hacker News thread.

We attracted $n = 352$ participants through this process from May 23, 2018 to August 17, 2020. To categorize them, despite anonymity, we rely on their self-reported background just as in the previous work (Section 2 for more information). Figures 1 through 11 describe our participants' backgrounds in detail.

The software developer group that this paper focuses on is demographically quite different from the scientific developer group considered in the previous paper. Most importantly, our recruitment process seems to have done a good job of capturing individuals from a more general software engineering background. Figure 1 shows the breakdown of the reported positions of the participants. Our participants are dominated by software engineers (53.7%), while the participants in the previous study were dominated by academics (61.3% were faculty or Ph.D. students).

[2]http://presciencelab.org/float

Position	n	%	Old %
Software Engineer	189	53.7	11.6
Undergraduate Student	37	10.5	3.5
Research Scientist	28	8.0	5.6
Ph.D. Student	25	7.1	36.7
M.S. Student	21	6.0	4.0
Faculty	18	5.1	24.6
Research Staff	6	1.7	8.5
Manager	6	1.7	1.5
Other	10	2.8	2.5

Figure 1: Positions of participants (software developer group). Old % is for the scientific developer group.

Area	n	%	Old %
Computer Science	190	54.0	40.2
Computer Engineering	43	12.2	9.5
Other Physical Science Field	38	10.8	19.1
Other Engineering Field	17	4.8	13.1
Electrical Engineering	15	4.3	4.5
Mathematics	14	4.0	5.0
Unreported	11	3.1	0.5
Statistics	4	1.1	0.5
Other Non-Physical Science Field	4	1.1	1.1
Self-Taught/Autodidact	3	0.9	-
None/No Formal Training	2	0.6	-
Aerospace Engineering	1	0.3	-
CE and CS	1	0.3	1.1
Applied Mathematics and CS	1	0.3	1.1
Information Technology	1	0.3	-
Game Development	1	0.3	-
Computational Chemistry	1	0.3	-
Physics, EE, Math, and Economics	1	0.3	-
Engineering Physics	1	0.3	-

Figure 2: Positions of participants (software developer group). Old % is for the scientific developer group.

Formal Training *in Floating Point*	n	%	Old %
None	115	32.7	26.1
At least one lecture within a course	112	31.8	31.2
One or more weeks within a course	63	17.9	24.6
One or more courses	55	15.6	17.6
Unreported	7	2.0	0.5

Figure 3: Formal Training *in floating point* of participants in the software developer group. Old % is for the scientific developer group.

The two groups are more similar when it comes to their reported areas, although computer science and engineering is more common in the software developer group. 2/3 of the software developer group report being in these areas. There is also a much smaller representation from other engineering fields or the physical sciences. Figure 2 provides the details of the breakdown by area.

The software developer group is slightly more likely than the scientific developer group to report having no formal training in floating point, or to have spent less time in formal training. A plurality of the software developers have no formal floating point training, while a plurality of the scientific developers report having one or more lectures of formal training in floating point. Figure 3 gives the breakdown. Note that from 1/4 to 1/3 of participants of

Informal Training *in Floating Point*	*n*	%	Old %
Read about it	272	77.3	68.3
Googled when necessary	212	60.2	69.4
Discussed with coworkers/managers/friends	124	35.2	44.7
Watched video	48	13.6	11.1
Took tutorial	30	8.5	9.0
Unreported	27	7.7	11.0
Trained by advisor or mentor	25	7.1	19.1

Figure 4: Informal Training *in floating point* of participants in the software developer group. Old % is for the scientific developer group. (Top 7 shown).

Software Development Role	*n*	%	Old %
My main role is as a software engineer	233	66.2	25.1
I develop software to support my main role	79	22.4	59.8
Unreported	17	4.8	2.5
I manage others who develop software to support my main role	14	4.0	9.5
My main role is to manage software engineers	9	2.6	3.0

Figure 5: Software Development Roles of participants in the software developer group. Old % is for the scientific developer group.

across the two groups received *no* formal training, which seems problematic given how common the use of floating point is among these developers, and more broadly.

Figure 4 breaks down both groups' reported informal training methods. Googling when necessary and reading about floating point remain the two most prevalent methods, although they swap between the two groups. Perhaps disturbingly, a software developer participant is 3 times less likely to report being trained by an advisor or mentor.

The software developers are 2.5 times more likely to report that their main role in software development is as a software engineer. These participant's roles contrast with the scientific developers, who are instead about 2.5 more likely to report that the develop software in support of their main role. Figure 5 details these differences and helps to support our claim that we have captured a distinctly different participant group in this second study.

Figure 6 breaks down the experience both groups report in languages that include floating point arithmetic. The three most common across both groups are C, C++ and Python, but the scientific developer group is considerably more likely to have used floating point in C, while the software developer group is more likely to have used it in Python. As might be expected, traditional languages for scientific computation are much less common in the software developer group.

Understanding of floating point arithmetic might be affected by the use of languages that provide arbitrary precision numerics. Figure 7 breaks down the two groups by their reported use of such languages and tools. Over 1/3 of both groups report no experience with such languages. The most common reported language is Mathematica. The differences between the groups are minor in this regard.

Floating Point Languages Experience	*n*	%	Old %
C	283	80.4	69.9
C++	251	71.3	68.3
Python	228	64.8	71.4
Java	181	51.4	50.3
Matlab	99	28.1	52.8
C#	76	21.6	13.1
Fortran	58	16.5	32.7
R	47	13.4	24.1
Perl	37	10.5	12.6
Haskell	29	8.2	6.0
Scheme/Racket	22	6.2	8.5
Javascript	14	4.0	3.0
Rust	11	3.1	2.0
ML	10	2.8	4.5
Unreported	10	2.8	0.5
Ruby	7	2.0	3.0

Figure 6: Floating Point Language Experience of participants in the software developer group. Old % is for the scientific developer group. 53 languages were reported. These had $n \geq 5$.

Arb. Precision Language Experience	*n*	%	Old %
Unreported	140	39.8	36.7
Mathematica	86	24.4	35.7
MPFR/GNU MultiPrecision Library	54	15.3	9.6
Other Language	46	13.1	10.0
Maple	40	11.4	14.6
Other Library	34	9.7	6.5
Scheme/Racket/LISP with BigNums	26	7.4	6.5
Haskell with arb. prec. and rationals	25	7.1	4.0
Matlab MultiPrecision Toolbox	14	4.0	5.0
Macsyma	7	2.0	2.5

Figure 7: Arbitrary Precision Language Experience of participants in the software developer group. Old % is for the scientific developer group. 32 languages/libraries were reported. These had $n \geq 5$.

Contributed Codebase Size	*n*	%	Old %
10,001 to 100,000 lines of code	139	39.5	32.7
1,001 to 10,000 lines of code	116	33.0	39.7
100,001 to 1,000,000 lines of code	52	14.8	8.5
>1,000,000 lines of code	18	5.1	4.5
100 to 1,000 lines of code	15	4.3	13.6
Unreported	11	3.1	0.5
<100 lines of code	1	0.3	0.5

Figure 8: Contributed Codebase Sizes of participants in the software developer group. Old % is for the scientific developer group.

We asked each participant to note the size of the largest codebase to which they have contributed, and then the extent of floating point usage in that codebase. Figures 8 and 9 detail the results. Not surprisingly, the software developer participants are more likely to have contributed to larger codebases. Additionally, they are slightly more likely to report that either floating point was incidental to the codebase, and slightly less likely to report that they or their team was involved in numerical correctness.

Contributed Codebase Floating Point Extent	*n*	%	Old %
FP incidental	158	44.9	38.7
FP intrinsic	92	26.1	31.7
FP intrinsic, I did numerical correctness	39	11.1	14.6
No FP involved	25	7.1	4.5
FP intrinsic, other team did numerical correctness	14	4.0	5.0
FP intrinsic, my team did numerical correctness	12	3.4	5.0
Unreported	12	3.4	0.5

Figure 9: Contributed Codebase Floating Point Extent of participants in the software developer group (within the codebase they built (Figure 8.) Old % is for the scientific developer group.

Involved Codebase Sizes	*n*	%	Old %
100,001 to 1,000,000 lines of code	105	29.8	18.1
>1,000,000 lines of code	97	27.6	18.1
10,001 to 100,000 lines of code	92	26.1	30.7
1,001 to 10,000 lines of code	38	10.8	26.6
Unreported	12	3.4	0.5
100 to 1,000 lines of code	6	1.7	4.0
<100 lines of code	2	0.6	1.0

Figure 10: Involved Codebase Sizes of participants in the software developer group. Old % is for the scientific developer group.

Involved Codebase Floating Point Extent	*n*	%	Old %
FP incidental	149	42.3	35.7
FP intrinsic	78	22.2	27.6
No FP involved	52	14.8	7.5
FP intrinsic, I did numerical correctness	22	6.2	11.6
FP intrinsic, other team did numerical correctness	19	5.4	8.5
FP intrinsic, my team did numerical correctness	17	4.8	6.5
Unreported	15	4.3	2.5

Figure 11: Involved Codebase Floating Point Extent of participants in the software developer group within the largest codebase they were involved with (Figure 10). Old % is for the scientific developer group.

Figures 10 and 11 present parallel questions to those just described, but here, we ask the participants to consider the largest codebase in which they have been involved in any capacity. Again, not surprisingly, the software developer participants were much more likely to be involved in very large codebases (almost 60% were involved in codebases bigger than 100,000 lines, compared to about 36% for the scientific developer group). Interestingly, however, the probability of floating point being intrinsic or incidental to those codebases was similar between the two groups.

Note that 189 of our participants identify specifically as having the position of software engineer (Figure 1). Because this is such a large number, we can treat this software engineer subgroup separately to allow us to draw an even sharper distinction from the scientific developer group of the initial study.

There are three takeaways from this discussion and data. First, we have clearly recruited a participant group that is quite distinct

Core Quiz				
# Correct (Δ old)	# Incorrect	# Don't Know	# No Answer	**# Chance**
10.0 (+1.5)	3.4	1.2	0.42	7.5
Optimization Quiz				
# Correct (Δ old)	# Incorrect	# Don't Know	# No Answer	**# Chance**
1.0 (+0.4)	0.3	**1.5**	0.2	2.0

Figure 12: Average (expected) performance of participants in the software developer group on the core and optimization quizzes. Δ old is the difference over the previous study of the scientific developer group.

from the participant group of the previous study. Second, the new participant group is arguably a good snapshot of software developers that we can compare with the previous study's snapshot of scientific developers. Finally, the software engineer subgroup allows us an even more sharply contrasting snapshot.

4 ANALYSIS RESULTS

We applied the same analysis methodology used in our previous study of scientific developer group to the software developer group (and its software engineer subgroup). Our goal was to address the same questions as in the previous work, namely:

- 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?

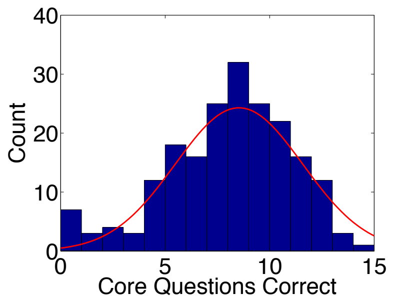
We now summarize the main results of our analysis of the software developers and how the groups differ.

It is important to note why our presentation is so rich in graphs and numbers. Although $n = 352$ is a large population, once it is *partitioned* for a typical factor, t-testing is not reasonable, as a t-test for a small n assumes the inputs are from a normal distribution, which is not an assumption that can be made here. Instead, we describe our interpretation and provide the data, given space constraints, so the reader can judge for themselves. In essence, we are providing an exploratory data analysis [22].

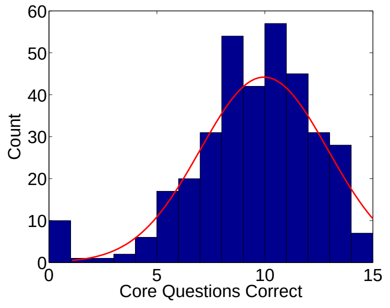
4.1 General understanding

Figure 12 shows the average (i.e. expected) scores of the software developers on the core and optimization quizzes, and shows the difference compared to the previous study of scientific developers. In both cases, participants generally feel they *can* answer the core quiz questions. However, the software developer group is considerably better at answering the questions correctly. The software developers perform well above chance (10/15 versus 8.5/15 for the scientific developers). The means of the scores of these two populations differ with $p < 0.01$.

Digging deeper, Figure 13 shows histograms of the two groups' scores on the core quiz. There is a subtlety here in that "Don't



(a) scientific developers (quoted from [7])



(b) software developers

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

Know" was a possible response to a question. The incidence of this, however, was < 15% for the core quiz in both studies. We can clearly see that the quiz produces a close-to-normal distribution for both groups, with the distribution shifted significantly toward better scores for the software developers. Still, the main news is that both groups are overconfident in their understanding of basic floating point behavior (which they likely saw in their training (Figures 3 and 4 for the software developer group).

In contrast, in the optimization quiz, participants in both groups generally recognize their ignorance, answering "Don't Know" from 1/2 to 2/3 of the time. Here, the software developer group does better as well, however. They are more likely to claim knowledge (although this is still rare), and when they do so are more likely to be correct. The reassuring point is that participants in both groups seem to be appropriately wary about compiler and hardware optimizations (perhaps these topics are less likely to be encountered in formal and informal training).

Figure 14 is a question-by-question breakdown of the core quiz. With two exceptions, Negative Zero and Square, the software developers outperform the scientific developers, in some cases quite significantly. Given how close the two groups are on Commutativity and Square, one possible interpretation is that these two questions are problematic.

Figure 15 is a question-by-question breakdown of the optimization quiz. While the participants in the scientific developer group from the earlier study asserted they did not know the answer over 50% of the time for all questions, the software developers did so only for the MADD and Flush to Zero questions. Their chance of getting a question correct was also higher for all questions. Note however, that in three of the four questions, the probability that the software developer participant either did not know, or answered incorrectly exceeds 2/3. The only question where answering correctly was ≥ 50% probable is Fast-math.

Question	% Correct	Δ old	% Incorrect	% Don't Know	% Unanswered
Commutativity	56.8	+3.5	31.0	10.2	2.0
Associativity	85.8	+16.5	8.2	3.7	2.3
Distributivity	90.1	+8.2	2.0	5.1	2.8
Ordering	90.3	+9.9	4.5	2.8	2.3
Identity	44.0	+27.4	50.0	3.4	2.6
Negative Zero	49.4	-9.4	41.5	6.5	2.6
Square	42.9	-4.3	42.0	12.8	2.3
Overflow	69.0	+8.2	20.4	6.8	3.7
Divide by Zero	20.5	+8.9	67.6	8.5	3.4
Zero Divide By Zero	75.6	+5.2	6.2	14.8	3.4
Saturation Plus	73.3	+18.5	18.2	5.7	2.8
Saturation Minus	71.0	+17.7	19.0	6.8	3.1
Denormal Precision	73.9	+21.6	14.2	8.8	3.1
Operation Precision	79.8	+6.4	7.7	9.6	2.8
Exception Signal	73.9	+4.6	9.4	13.6	3.1

Figure 14: Performance of the software developergroup on 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. Δ old is the percentage point difference compared to the scientific developer group.

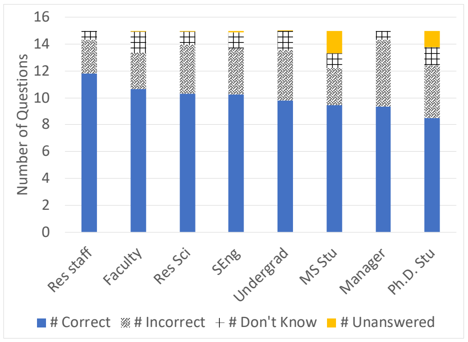


Figure 16: Effect of Position on core quiz scores.

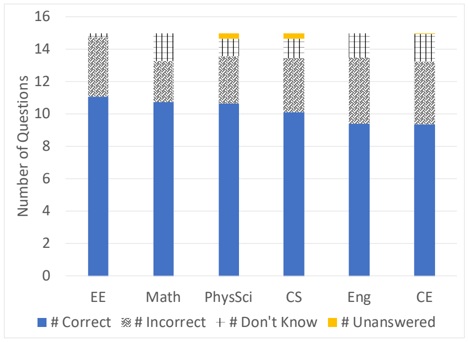
4.2 Factor analysis for core quiz

We considered the effect of each of our background factors on the participant score in the core quiz. Each factor was considered in isolation, and we focus on the largest effects and contrasts with the scientific developer group.

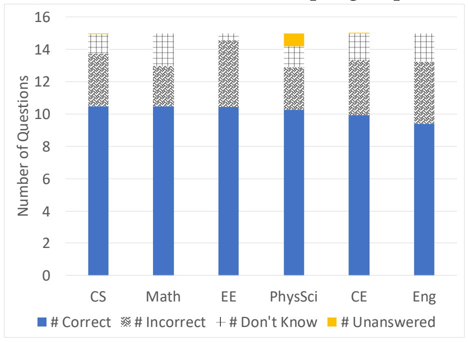
In contrast to previous study of the scientific developer group, where code size, namely the size of the largest codebase the participant had contributed to or been involved with, had the most marked effect, we find hardly any effect in the software developer group. Even if we focus on the software engineer subgroup, we see virtually no effect. A possible explanation is that a software developer is more likely to have worked on more numerous codebases, and thus, even if those codebases were small, they still may have had the training benefit.

Question	% Correct	Δ old	% Incorrect	% Don't Know	% Unanswered
MADD	26.7	+11.1	16.8	50.8	5.7
Flush to Zero	20.7	+7.1	13.4	60.2	5.7
Standard-compliant Level	20.7	+12.2	29.0	44.6	5.7
Fast-math	56.5	+27.4	2.3	35.2	6.0

Figure 15: Performance of the software developer group on the Optimization quiz questions. The highlighted questions were reported as unknown by more than half of the participants. Δ old is the percentage point difference compared to the scientific developer group.



(a) entire software developer group



(b) software engineer subgroup only

Figure 17: Effect of Area on core quiz scores.

The most significant factor we find in the software developer group's performance is Position, which shows a variation of 4/15. This variation is the same as the largest variation seen in the previous study of the scientific developer group (for the code size factor). Figure 16 shows the breakdown.[3]

For the scientific developer group, the second most significant factor was the Area of the participants. This remains the case for the software developer group, with the variation being about 2/15, which is markedly smaller than the 3.5/15 variation reported for the scientific developers. Figure 17 shows the breakdown. Interestingly, if we consider the software engineers only, this factor has much less effect.

In the previous study, the third most significant factor we encountered was Software Development Role. In contrast, for the software developer group, we find this to have no effect, except

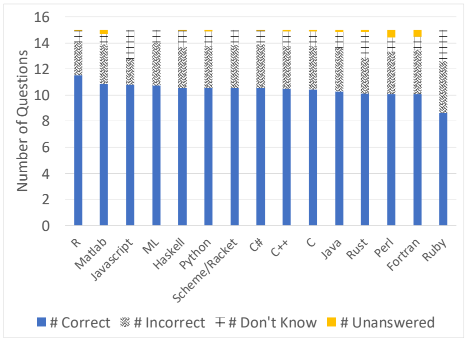


Figure 18: Effect of Languages Used on core quiz scores of the software engineer subgroup.

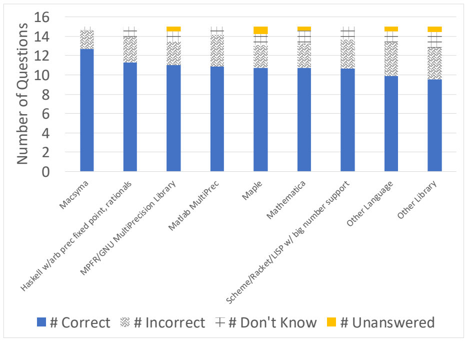


Figure 19: Effect of Arbitrary Precision Languages Used on core quiz scores of the software engineer subgroup.

when the participant did not specify their development role, in which case the average correctness score drops by 4/15.

Similarly, while the previous study showed that Formal Training (in floating point) had a small effect (2/15), the effect is much smaller here (1/15), and shrinks even more when the software engineer subgroup is considered. Sadly, formal training about floating point behavior does not seem to be effective. A possible explanation is that given when this is taught in a CS sequence (typically in an introductory computer systems course in the sophomore year), the participants are not really in a position to leverage what they learn or to have it reinforced.

Interestingly, among the software engineer subgroup, Languages Used, and Arbitrary Precision languages used seem to have a high apparent effect. The effect variations are 3.5/15, and 3/15, respectively. In contrast, for the previous study's scientific developer group, the effects were either minimal or ambiguous. Figures 18, and 19 show the breakdown.

[3]We use the same presentation format for this data as in the previous study to support straightforward comparisons between the two papers. For each value the factor can take on, we provide a stacked bar that shows the average count of correctly answered questions, incorrectly answered questions, questions answered "Don't Know", and questions that went unanswered. The bars, and hence the values are sorted by descending order of average number of correctly answered questions.

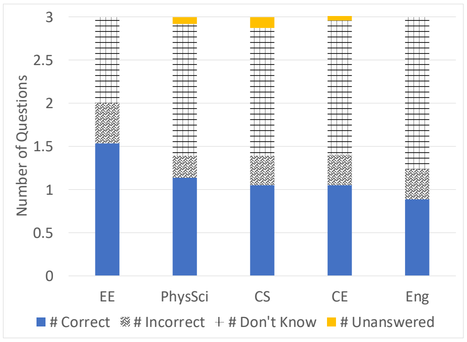


Figure 20: Effect of Area on optimization quiz scores.

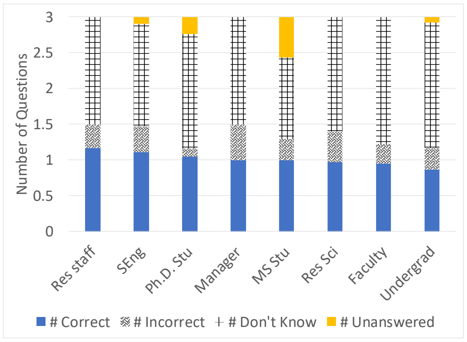


Figure 21: Effect of Position on optimization quiz scores.

4.3 Factor analysis for optimization quiz

The main story about the optimization quiz for the software developer group compared to the scientific developer group is the continued dominance of the response "Don't Know" regardless of how the data is sliced by the factors.

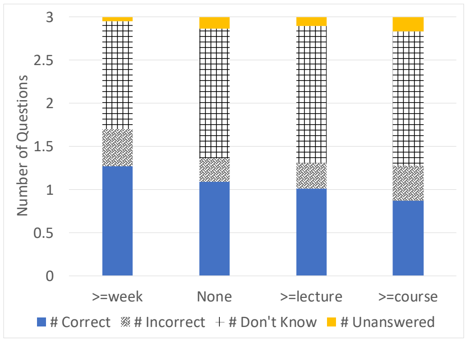
Unlike the previous scientific developer group, where Software Development Role was the most significant factor, this factor has very little effect within the software developer group. Indeed the variation is almost zero except for responses where the participants did not specify a role.

Area, which in the previous study had the second highest impact, here has the highest. However, the variation we see here is reduced to 0.5/3 which is a lower than the 0.7/3 we saw with the scientific developer group. Figure 20 gives a breakdown.

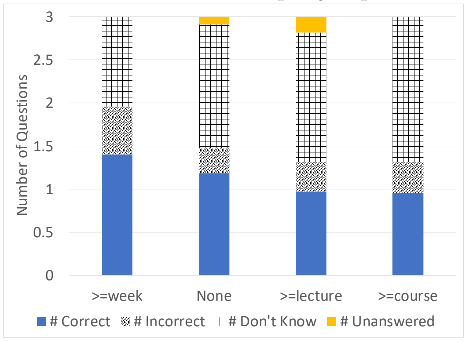
Unlike the scientific developer group, Position plays a secondary role with the software developer group, although the effect is tiny as can be seen in Figure 21's breakdown.

Quite unlike with the previous scientific developer group, the effect of Formal Training (in floating point) is considerable in the software developer group, and this effect is larger in the software engineer subgroup. Figure 22 gives a breakdown. Overall, the effect's strength is similar to that of Position, although it is stronger for the subgroup.

Interestingly, among the software engineer subgroup, Informal Training in Floating Point, and Arbitrary Precision languages used seem to have a high apparent effect. The effect variations are 1.25/3 in both cases. In contrast, for the previous study's scientific developer group, the effects were either minimal or ambiguous. Figures 23, and 24 show the breakdowns.



(a) software developer group



(b) software engineer subgroup only

Figure 22: Effect of Formal Training on optimization quiz scores.

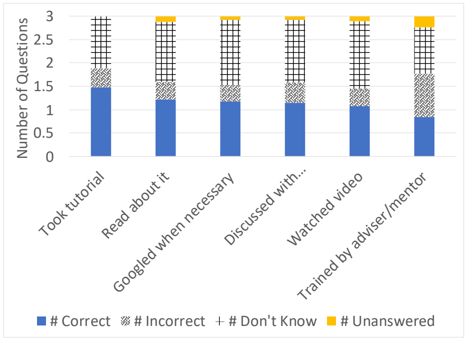
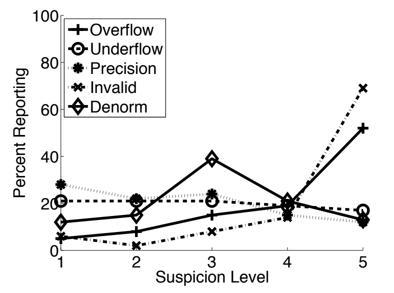


Figure 23: Effect of Informal Training on optimization quiz scores for the software engineer subgroup.

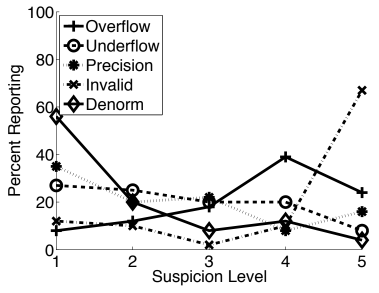
4.4 Suspicion analysis

As described earlier, some exceptional conditions that the hardware can detect are more problematic than others. We would hope that the developer is more suspicious of NaNs (Invalids) than of Infinities (Overflows), and more suspicious of Infinities than of other conditions such as Underflows, Denorms, and Precision (rounding).

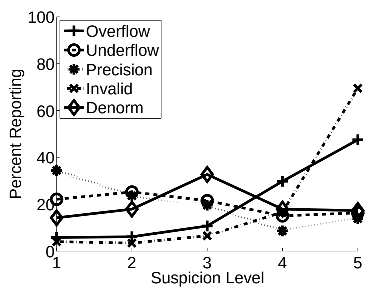
Figure 25 shows the distribution of reported suspicion with (a) the scientific developer group from the previous paper, (b) the student group from the previous paper, and (c) the software developer group of this paper. Note that all three groups are generally more suspicious of Invalids and Overflows than of the other conditions. Both the scientific developer and software developer groups find Overflow more suspicious than the students. However, it is also the



(a) scientific developer group ($n = 199$)



(b) student group ($n = 52$)



(c) software developer group ($n = 352$)

Figure 25: Distribution of suspicion for different exceptional conditions. (a) and (b) are quoted from [7].

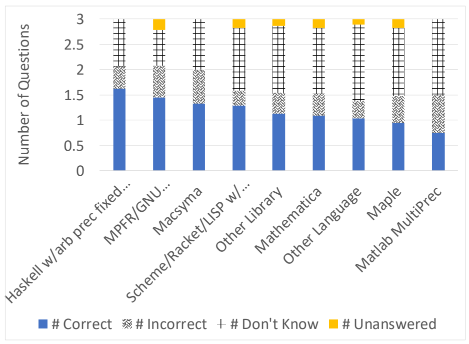


Figure 24: Effect of Arbitrary Precision Languages Used on optimization quiz scores for the software engineer subgroup.

Formal Training *in Floating Point* of Top 10% of Respondents	*n*	%	Old %
None	12	34.3	30.0
One or more weeks within a course	9	25.7	25.0
At least one lecture within a course	9	25.7	25.0
One or more courses	5	14.3	20.0

Figure 26: Formal Training *in floating point* of the top 10% of core quiz participants. Old % is for the scientific developer group.

case that, across all three groups, about 1/3 of the participants do not report the maximum possible suspicion of a computation that somewhere encountered a NaN.

4.5 Commonalities among top scorers

We now consider just the top 10% of scorers in the core quiz from both the scientific developer and software developer groups, 20 and 35 participants, respectively. As shown in Figure 26, there seems to be little relationship to formal training. Indeed, about 1/3 reported none at all. Most top scoring participants report some form of informal training, as indicated in Figure 27, but very few top scorers took tutorials or were trained by a mentor, which is surprising.

Experience in C or C++ seems to be a commonality for top scorers as seen in Figure 28, although Python and other languages are also represented. Experience with traditional scientific computing languages like Fortran and Matlab is not that common, particularly among the top-scoring software developers.

Informal Training *in Floating Point* of Top 10% of Respondents	*n*	%	Old %
Read about it	31	88.6	75.0
Googled when necessary	22	62.8	55.0
Discussed with coworkers/managers/friends	13	37.1	40.0
Watched video	5	14.3	15.0
Took tutorial	4	11.4	10.0
Unreported	2	5.7	10.0
Trained by advisor or mentor	2	5.7	25.0
Other	3	8.6	10.0

Figure 27: Informal Training *in floating point* of the top 10% of core quiz participants. Old % is for the scientific developer group. (Top 8 shown).

Floating Point Languages Experience of Top 10% of Respondents	*n*	%	Old %
C	29	82.9	90.0
C++	27	77.1	95.0
Python	26	74.3	70.0
Java	20	57.1	60.0
Matlab	11	31.4	60.0
C#	7	20.0	20.0
Fortran	6	17.1	35.0

Figure 28: Floating Point Language Experience of the top 10% of core quiz participants. Old % is for the scientific developer group. 53 languages were reported. These had $n \geq 5$.

Arb. Precision Language Experience of Top 10% of Respondents	*n*	%	Old %
Mathematica	15	42.8	55.0
Unreported	8	22.8	10.0
Other Language	8	22.8	15.0
Maple	8	22.8	20.0
MPFR/GNU MultiPrecision Library	6	17.1	15.0

Figure 29: Arbitrary Precision Language Experience of the top 10% of core quiz participants. Old % is for scientific developer group.

The top scorers in both groups are more likely to have had experience with an arbitrary precision language or library, as seen in Figure 29. Mathematica, in particular, was used by almost half of the top scorers in the software developer group and over half of the top scorers in the scientific developer group.

Lastly, the top scorers in both the software developer and the scientific developer groups tend to work on larger codebases, with well over half of participants in both groups having built codebases with over ten thousand lines of code.

5 CONCLUSIONS

The study we reported in this paper suggests that software developers generally have a better understanding the vagaries of the IEEE floating point standard and its implementations than scientific developers. The two groups are similarly suspicious of exceptional conditions, arguably insufficiently so. The software developer group still has plenty of room for improvement, however, and the observations and suggested actions that conclude the previous work [7] hold for them as well.

It is alarming that that a group (software developers) with *less* formal training about floating point performs slightly *better* than a group (scientific developers) with more. But the bigger picture issue is that formal training seems to have only a small effect on understanding. This suggests that pedagogy in this area is an important task that the HPC community or others should revive. Perhaps the systems community has not yet found the correct training methods or the relevance of IEEE 754 vagaries is not being properly demonstrated to students.

Furthermore, these results suggest that the systems and HPC communities need to make further effort to impart proper suspicion of floating point onto developers. Because more of the software developer group reported no formal floating point training, perhaps it would pay to modify commonly employed tools (compilers, linters, etc.) to warn users of potentially problematic behavior. One approach to doing this might be to integrate the increasing set of tools for finding floating point issues, for example to produce warnings during the ordinary compilation process.

Lastly, perhaps the data suggests that the community is in need of a number system with fewer idiosyncrasies [4], or at least a mechanism to allow the number system used to be selectable at compile- or run-time.

REFERENCES

[1] Tao Bao and Xiangyu Zhang. 2013. On-the-fly Detection of Instability Problems in Floating-point Program Execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*

[2] Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Holger E. Jones. 2019. Multi-level Analysis of Compiler-Induced Variability and Performance Tradeoffs. In *Proceedings of the 28th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2019).*

[3] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. 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).*

[4] Hans-J. Boehm. 2020. Towards an API for the Real Numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[5] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).* 300–315.

[6] Peter Dinda, Alex Bernat, and Conor Hetland. 2020. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications. In *Proceedings of the 29th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2020).*

[7] Peter Dinda and Conor Hetland. 2018. Do Developers Understand IEEE Floating Point?. In *Proceedings of the* 32^{nd} *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018).*

[8] C. Halverson, C. Swart, J. Brezin, John T. Richards, and C. Danis. 2008. Towards an Ecologically Valid Study of Programmer Behavior for Scientific Computing. In *Proceedings of the 1st International Workshop on Software Engineering for Computational Science and Engineering (SECSE).*

[9] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software?. In *Proceedings of the 2nd International Workshop on Software Engineering for Computational Science and Engineering (SECSE).* 1–8.

[10] D. F. Kelly. 2007. A Software Chasm: Software Engineering and Scientific Computing. *IEEE Software* 24, 06 (nov 2007), 120, 118–119. https://doi.org/10.1109/MS.2007.155

[11] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. 2013. Dynamic floating-point cancellation detection. *Parallel Comput.* 39, 3 (2013), 146–155.

[12] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. 2015. RAIVE: Runtime Assessment of Floating-point Instability by Vectorization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[13] Yang Li, Nitesh Narayan, Jonas Helming, and Maximilian Koegel. 2011. A domain specific requirements model for scientific computing. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE).* 848–851. https://doi.org/10.1145/1985793.1985922

[14] Erika S. Mesh. 2015. Supporting Scientific SE Process Improvement. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, Vol. 2. 923–926. https://doi.org/10.1109/ICSE.2015.293

[15] Reed Milewicz, Gustavo Pinto, and Paige Rodeghero. 2019. Characterizing the Roles of Contributors in Open-Source Scientific Software Projects. In *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR).* 421–432.

[16] Daniel Milroy, Allison Baker, Dorit Hammerling, and Youngsung Kim. 2019. Making Root Cause Analysis Feasible for Large Code Bases: A Solution Approach for a Climate Model. In *Proceedings of the 28th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2019).*

[17] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[18] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[19] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *Proceedings of the 2017 IEEE International Symposium on Workload Characterization (IISWC).* 229–238.

[20] Victoria Stodden, Peixuan Guo, and Zhaokun Ma. 2013. Toward Reproducible Computational Research: An Empirical Analysis of Data and Code Policy Adoption by Journals. *PloS One* 8, 6 (2013).

[21] Victoria Stodden, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P.A. Ioannidis, and Michela Taufer. 2016. Enhancing reproducibility for computational methods. *Science* 354, 6317 (December 2016).

[22] John Tukey. 1977. *Exploratory Data Analysis.* Pearson.