Do You Understand IEEE Floating Point?

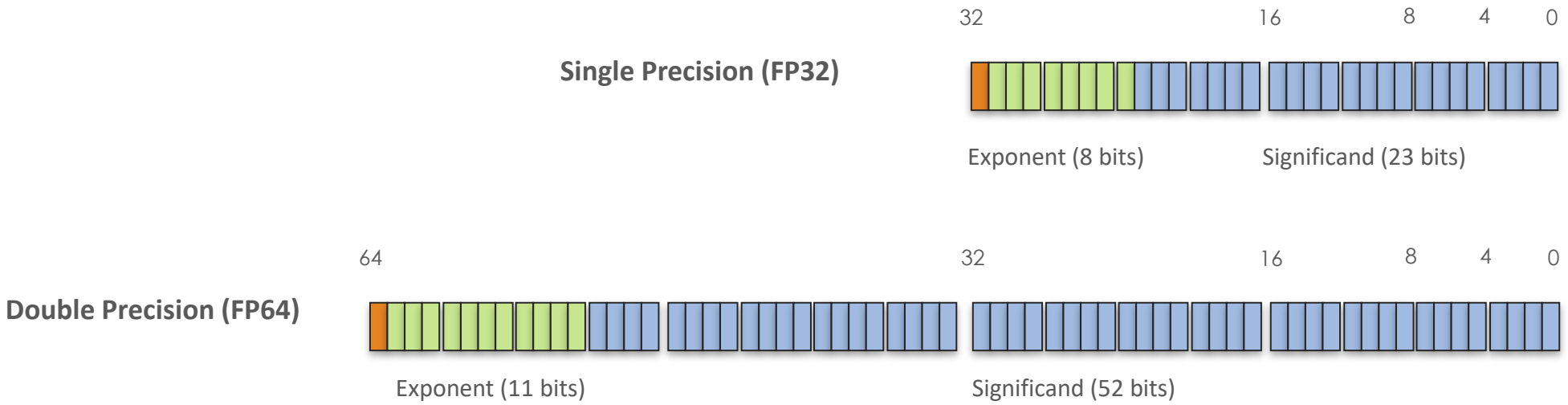
JAMES MADISON UNIVERSITY®

Do You Understand IEEE Floating Point?

(No.)

JAMES MADISON UNIVERSITY®

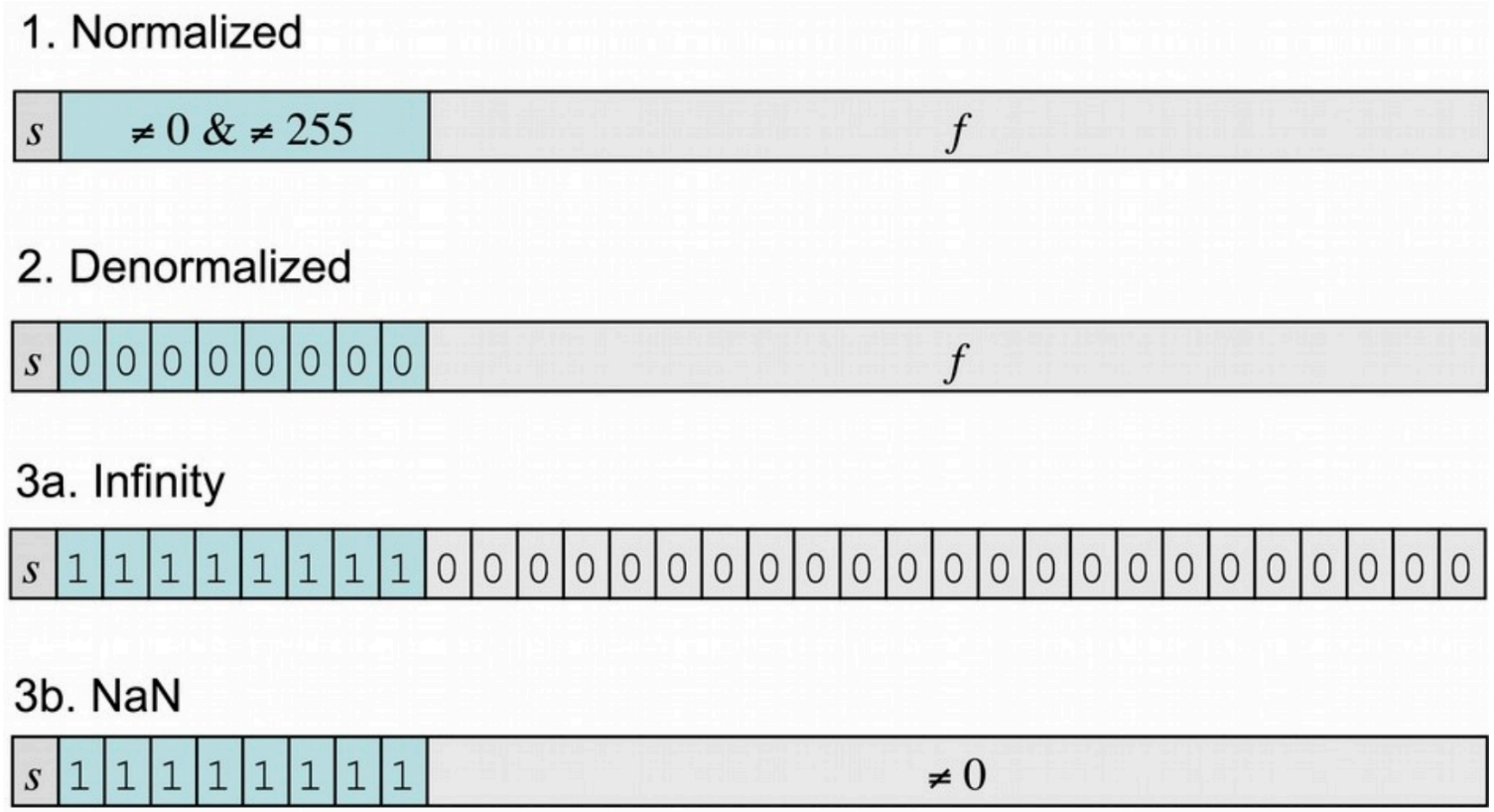
IEEE Floating Point (CS 261 review)



$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{significand} \times 2^{\text{exponent}}$$

0.significand for denormal numbers

IEEE Floating Point (CS 261 review)



IEEE Floating Point (CS 261 review)

Denormal numbers provide gradual underflow near zero

Offset/bias encoding of exponent retains integer sorting order

		Exponent			Fraction		Value		
Description	Bit representation	e	E	2^E	f	M	$2^E \times M$	V	Decimal
Zero	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest positive	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denormalized	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest normalized	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest normalized	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	0 1111 000	—	—	—	—	—	—	∞	—

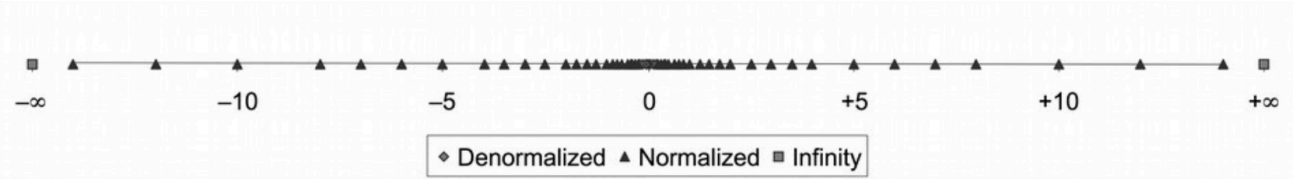
Figure 2.35 Example nonnegative values for 8-bit floating-point format. There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

IEEE Issues

- Rounding error
- Cancellation
- Exceptions
- Not-A-Number (NaN)
- Non-uniform distribution
- Positive and negative zero
- Denormalized numbers
- Compiler optimizations
- Fused multiply-add (FMA) instructions

```
  2.491264  (7)      1.613647  (7)
- 2.491252  (7)    - 1.613647  (7)
  0.000012  (2)      0.000000  (0)

(5 digits cancelled)   (all digits cancelled)
```



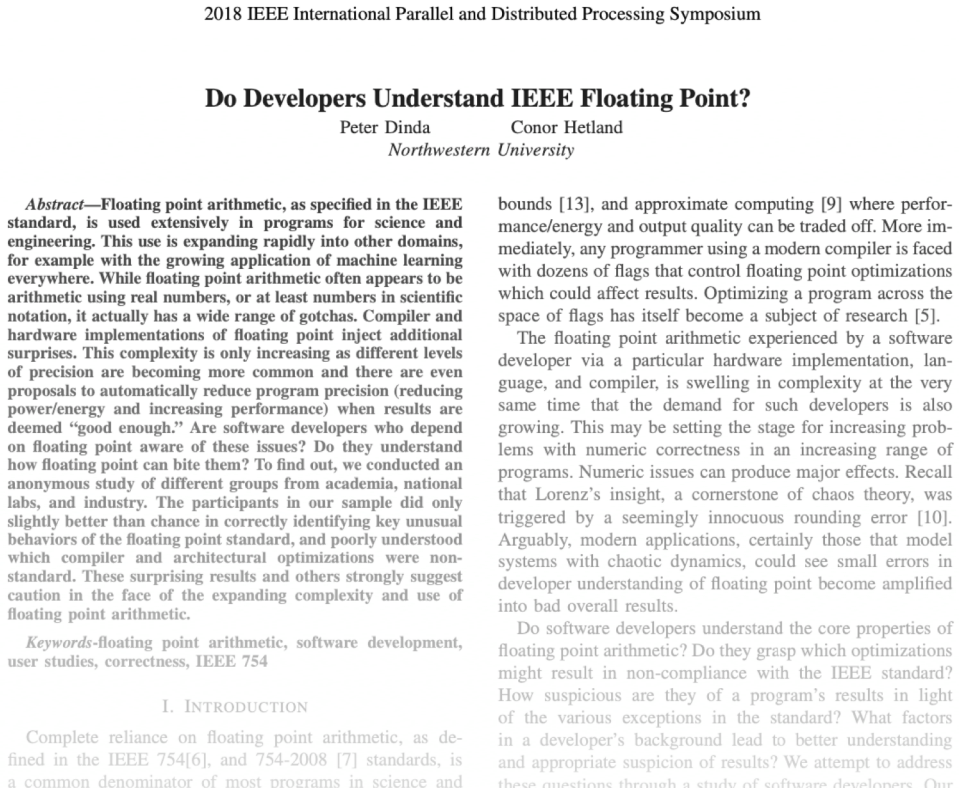
youtube.com/watch?v=9hdFG2GcNuA

Research Questions [Dinda 2018]

- 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?

Study Survey Design [Dinda 2018]

- Background
  - Current position and area
  - Training re: floating point
  - Software development experience
- Core knowledge quiz
- Optimization quiz
- “Suspicion” quiz



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 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

doi.org/10.1109/IPDPS.2018.00068

Core Quiz: Always true in IEEE arithmetic?

((a + b) – a) = b	a = 10^{20}, b = 1 *(rounding)*
(a + b) + c = a + (b + c)	a = 10^{20}, b = -a, c = 1
a(b + c) = ab + ac	a = 10^{20}, b = 1+10^{20}, c = 1-10^{20}
(a + 1) ≠ a	a = 10^{20} or 1/0 *(infinity)*
a = a	a = 0/0 *(NaN)*
+0 = -0	YES!
$a^2 > 0$	Yes, for non-NaNs (not true of signed ints!)

Optimization Quiz

- Can the common "`--fast-math`" compiler optimization result in non-standard-compliant behavior?
  - YES!
- Is the more efficient "fused multiply-add" (FMA) instruction part of the IEEE standard (i.e., mandated on all implementations)?
  - In newer (2008) standard but not the original
- What compiler level of optimization is generally the highest possible that preserves IEEE standard-compliant behavior?
  - -O2
- Is the flush-to-zero (FTZ) processor flag that eliminates denormalized numbers part of the IEEE standard?
  - NO! (but "on" by default in some hardware)

"Suspicion" Quiz: How "bad" is an IEEE exception?

- Overflow: result was infinity
  - Usually a sign of a bug
- Underflow: result was zero
  - Usually ok
- Precision: result required rounding
  - Usually ok, but can be a problem
- Invalid: result was NaN
  - Usually a sign of a bug
- Denorm: result was denormalized
  - Usually ok

- Answer choices:
  - Usually ok
  - Usually ok, but can be a problem
  - Usually a sign of a bug

Participant Recruitment

"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] 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." (Section III)

Results

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 $n \geq 5$.

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.

Results

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).

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.

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

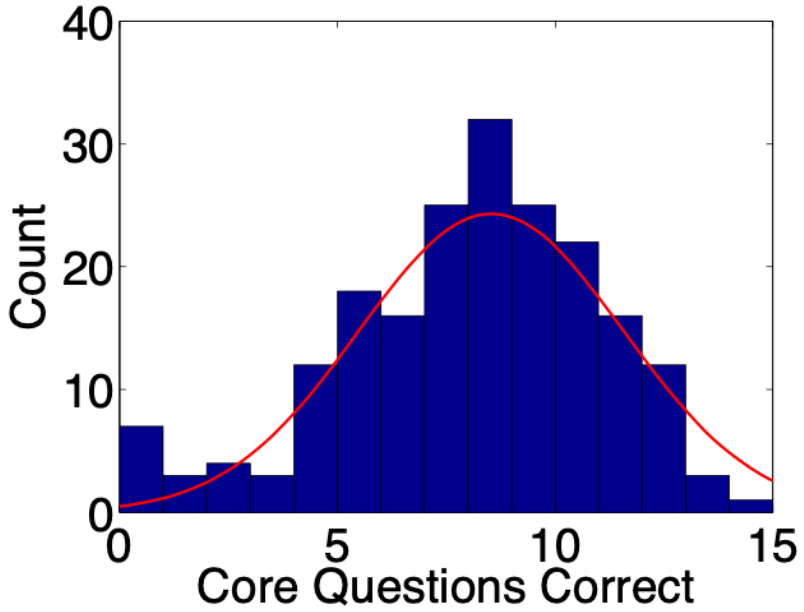
Figure 11: Involved Codebase Floating Point Extent of participants within the largest codebase they were involved with (Figure 10).

Results

"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. …

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." (Section III)

Results



Mean = 8.5

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

Results

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.

Results

"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 participants generally recognized their ignorance, answering 'Don't Know' **over 2/3 of the time.**" (Section IV.A)

Results

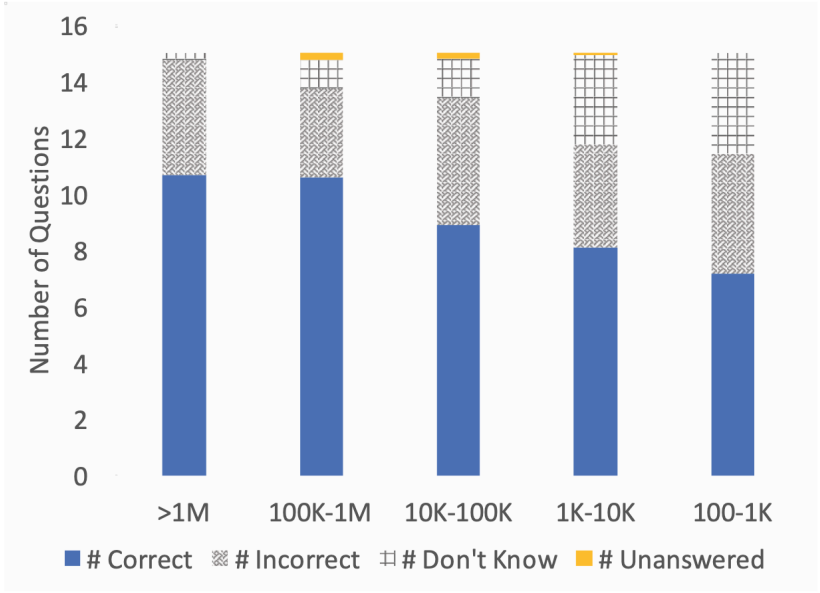


Figure 16: Effect of Contributed Codebase Size on core quiz scores.

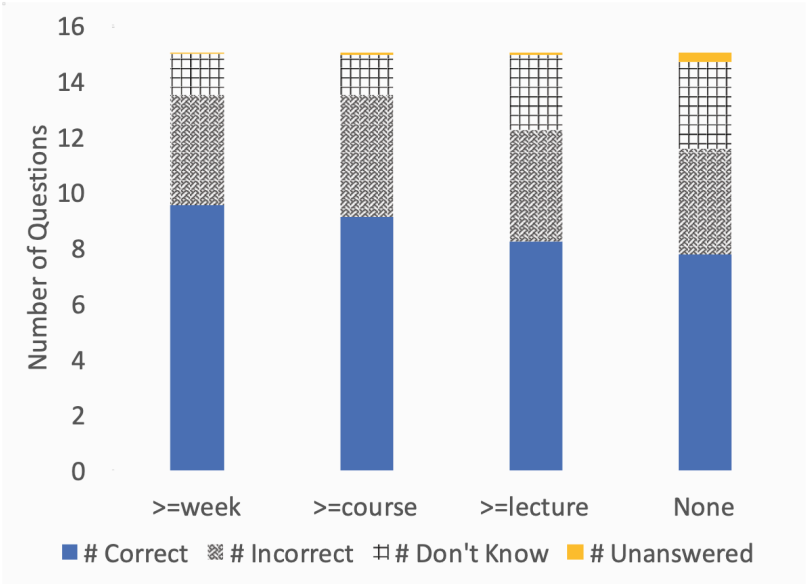
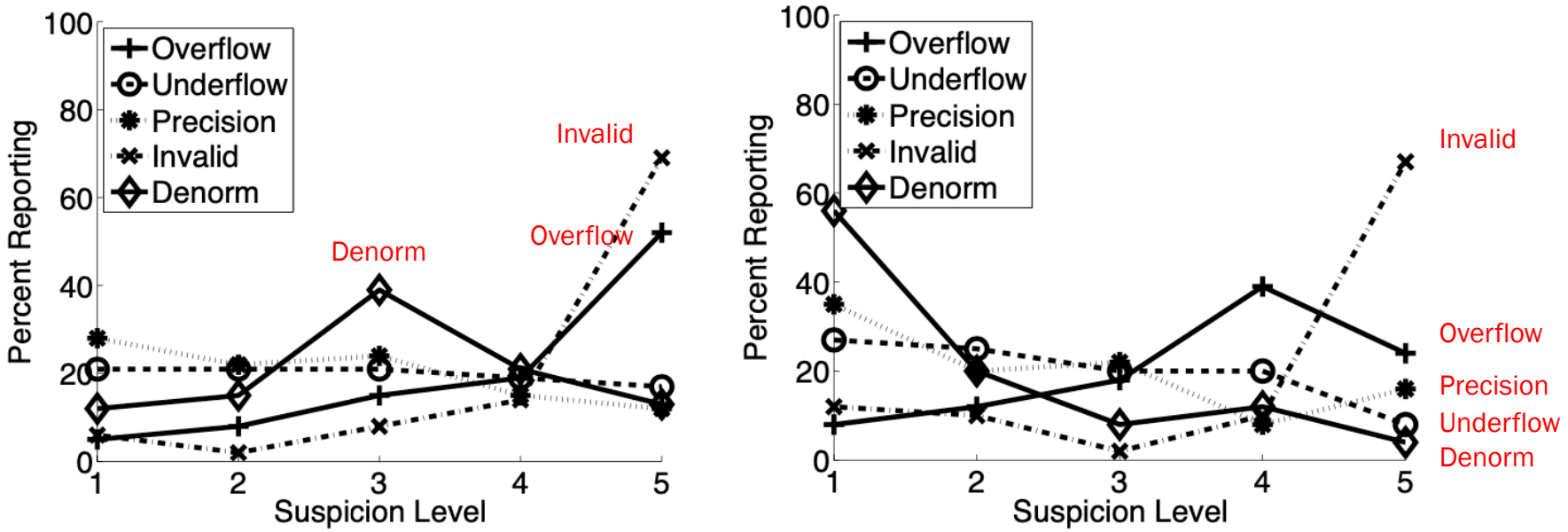


Figure 19: Effect of Formal Training (in floating point) on core quiz scores.

Results



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

Figure 22: Distribution of suspicion for different exceptional conditions.

Conclusions

- Observation: Many developers do not understand core floating point behavior well, **yet they believe they do**
- Observation: Many developers recognize their lack of knowledge about floating-point optimization
- Action: Develop new educational processes (e.g., with SIGHPC)
- Action: Disable floating-point optimizations by default
- Action: Develop/improve analysis tools
  - Local research projects: **CRAFT**, **ADAPT**, **FloatSmith**
- Action: Improve access to arbitrary precision implementations
  - Local research project: **SHVAL**

Thanks! (and good luck!) ☺

If you're interested, talk to me or visit my website: `w3.cs.jmu.edu/lam2mo`