Verifying Bit-Manipulations of Floating-Point

Wonyeol Lee Rahul Sharma Alex Aiken
Stanford University

This Talk

- Example:

$$e^{x}$$

mathematical specification

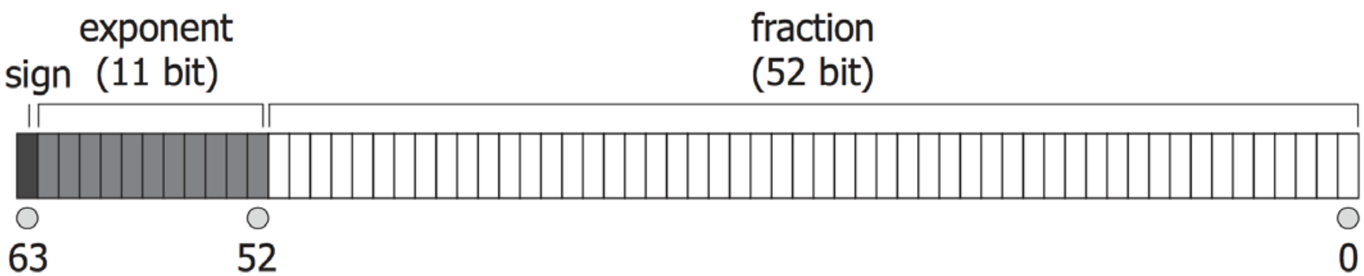
$\longleftrightarrow$ how different?

```
...
vpslld   $20,   %xmm3, %xmm3
vpshufd  $114,  %xmm3, %xmm3
vmulpd   C1,    %xmm2, %xmm1
vmulpd   C2,    %xmm2, %xmm2
...
```

floating-point implementation

- Goal: Bound the difference between spec and implementation
- Key contribution: Verify binaries that mix floating-point and bit-level operations
  - Intel's implementations of transcendental functions

Floating-Point Numbers



- Example: $\underbrace{1}\ \underbrace{01111111111}\ \underbrace{1100\cdots00}\ _{(2)}$

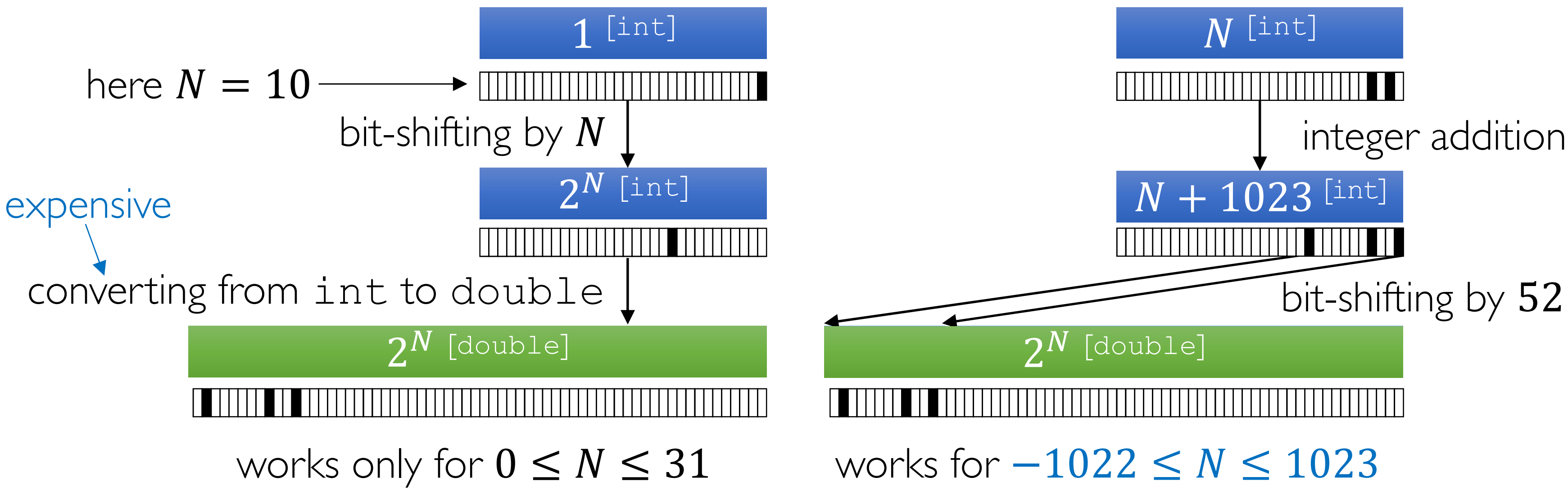
  $= (-1)^{\boxed{1}} \cdot 2^{\boxed{1023}-1023} \cdot 1.\boxed{110\cdots00}_{(2)}$

- Automatic reasoning about floating-point is not easy
  - have rounding errors
  - don't obey some algebraic rules of real numbers
  - Associativity: $1 + (10^{30} - 10^{30}) = 1 \neq 0 = (1 + 10^{30}) - 10^{30}$

- It becomes much harder if bit-level operations are used

Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`)



- Such bit-manipulations are ubiquitous in highly optimized floating-point implementations

- If a code mixes floating-point and bit-level operations, reasoning about the code is difficult

Problem Statement

e^x

mathematical specification

$f: \mathbb{R} \to \mathbb{R}$

$[-1, 1]$

input range $X \subseteq \mathbb{R}$

```
...
vpslld   $20,   %xmm3, %xmm3
vpshufd  $114,  %xmm3, %xmm3
vmulpd   C1,    %xmm2, %xmm1
vmulpd   C2,    %xmm2, %xmm2
...
```

binary P that mixes floating-point and bit-level operations

- Goal: Find a small $\Theta > 0$ such that

$$\left|\frac{f(x) - P(x)}{f(x)}\right| \leq \Theta \text{ for all } x \in X$$

  - i.e., prove a bound on the maximum precision loss

Possible Alternatives

- Exhaustive testing
  - feasible for 32-bit float: $\sim \mathbf{30}$ seconds (with $\mathbf{1}$ core for `sinf`)
  - infeasible for 64-bit double: $> \mathbf{4000}$ years $(= \mathbf{30}$ seconds $\times\, \mathbf{2}^{\mathbf{32}})$
  - infeasible even for input range $\mathbf{X} = [-1, 1]$
    $\because$ (# of doubles between $-\mathbf{1}$ and $\mathbf{1}$) $= \frac{1}{2}$ (# of all doubles)

- Machine-checkable proofs
  - Harrison used HOL Light to prove Intel's transcendental functions are very accurate [FMCAD'00]
  - "The construction of these proofs often requires considerable persistence." [FMSD'00]

Possible Automatic Alternatives

- If only floating-point operations are used, various automatic techniques can be applied
  - e.g., Astree [PLDI'03], Fluctuat [FMICS'09], ROSA [POPL'14], FPTaylor [FM'15]

- Several commercial tools (e.g., Astree, Fluctuat) can handle certain bit-trick routines

- We are unaware of a general technique for verifying mixed floating-point and bit-level code

Our Method

e^x Explained

```
vmovddup   %xmm0,  %xmm0
vmulpd     L2E,    %xmm0,  %xmm2
vroundpd   $0,     %xmm2,  %xmm2
vcvtpd2dqx %xmm2,  %xmm3
vpaddd     B,      %xmm3,  %xmm3
vpslld     $20,    %xmm3,  %xmm3
vpshufd    $114,   %xmm3,  %xmm3
vmulpd     C1,     %xmm2,  %xmm1
vmulpd     C2,     %xmm2,  %xmm2
vaddpd     %xmm1,  %xmm0,  %xmm1
vaddpd     %xmm2,  %xmm1,  %xmm1
vmovapd    T1,     %xmm0
vmulpd     T12,    %xmm1,  %xmm2
vaddpd     T11,    %xmm2,  %xmm2
...
vaddpd     %xmm0,  %xmm1,  %xmm0
vmulpd     %xmm3,  %xmm0,  %xmm0
retq
```

x

$N = \text{round}(x \cdot \log_2 e)$

2^N

$r = x - N \cdot \ln 2$

$$e^r \approx \sum_{i=0}^{12} \frac{r^i}{i!}$$

$$e^x = e^{N \cdot \ln 2} \cdot e^r \approx 2^N \cdot e^r$$

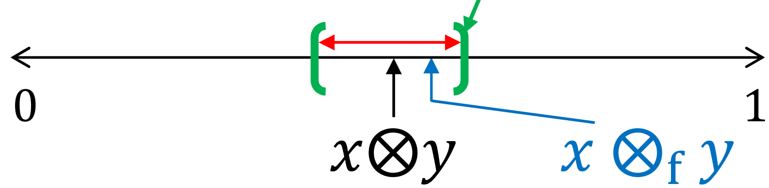
Goal: Find a small $\Theta > 0$ such that

$$\left|\frac{e^x - 2^N e^r}{e^x}\right| \leq \Theta \text{ for all } x \in X$$

1) Abstract Floating-Point Operations

- Assume only floating-point operations are used
- $(1+\epsilon)$ property
  - A standard way to model rounding errors

$$x \otimes_{\mathrm{f}} y \in \boxed{\{(x \otimes y)(1+\delta) : |\delta| < \epsilon\}}$$



  - For 64-bit doubles, $\epsilon = 2^{-53}$
  - This property has been used in previous automatic techniques (FPTaylor, ROSA, …) for verifying floating-point programs

1) Abstract Floating-Point Operations

- Compute a symbolic abstraction $A_{\vec{\delta}}(x)$ of a program P
  - Example:

$$A_{\vec{\delta}}(x) = \big((2 \times x)(1 + \delta_1) + 3\big)(1 + \delta_2)$$

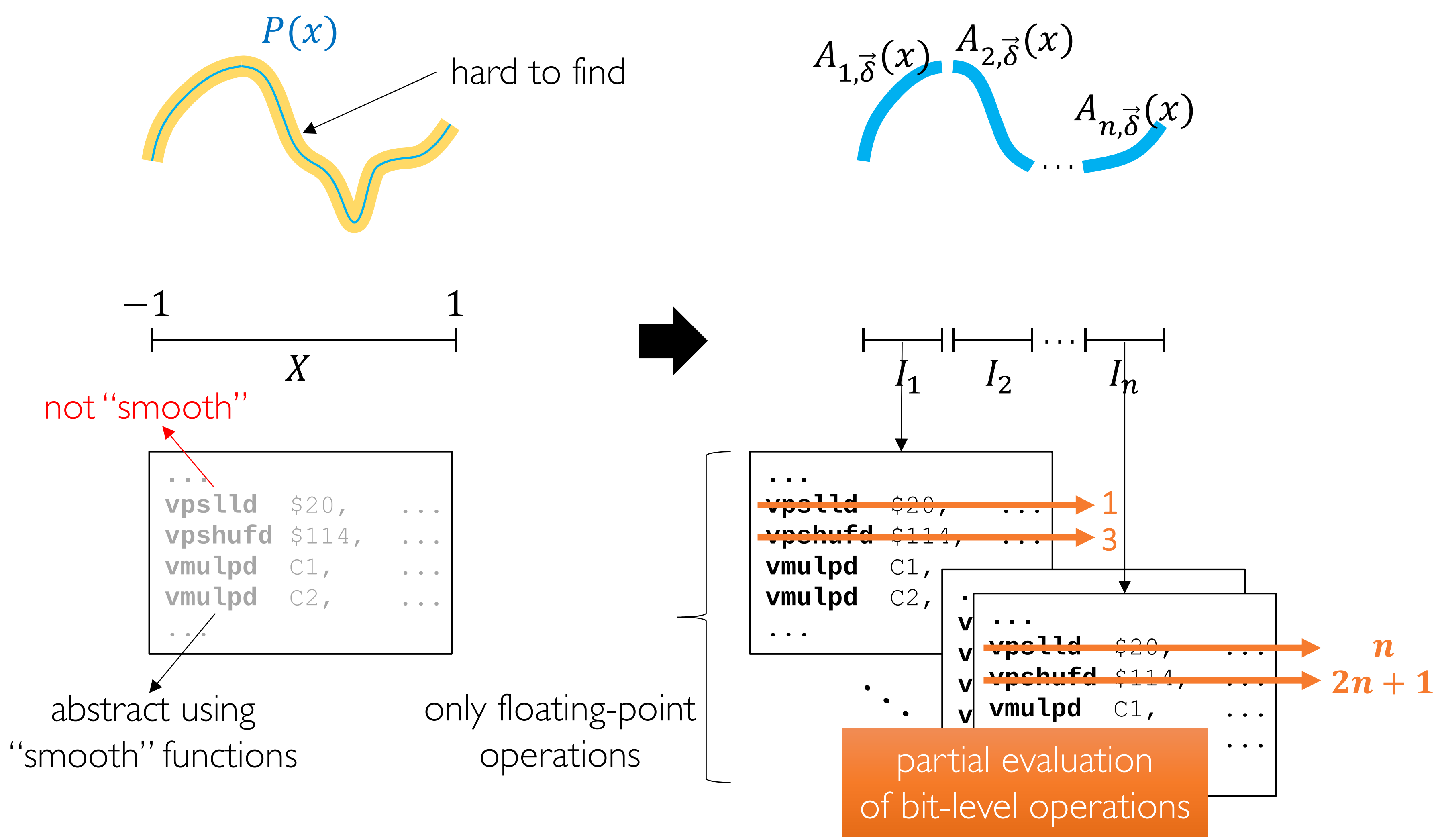
- From $(1 + \epsilon)$ property, $A_{\vec{\delta}}(x)$ satisfies

$$P(x) \in \left\{A_{\vec{\delta}}(x) : |\delta_i| < \epsilon\right\} \text{ for all } x$$

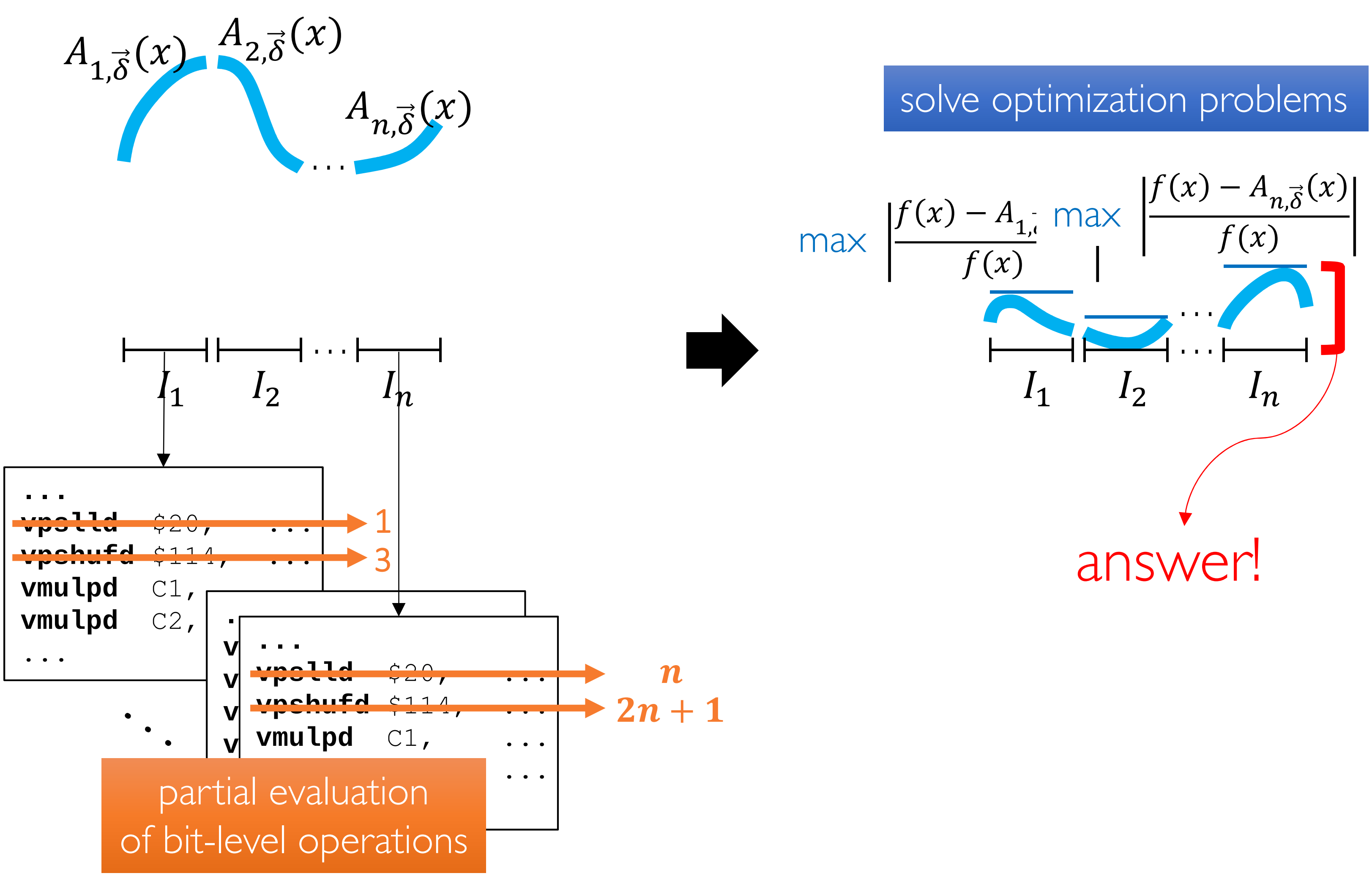
  - Example:

$$P(x) \in \left\{\big((2 \times x)(1 + \delta_1) + 3\big)(1 + \delta_2) : |\delta_1|, |\delta_2| < \epsilon\right\}$$

Our Method: Overview



Our Method: Overview

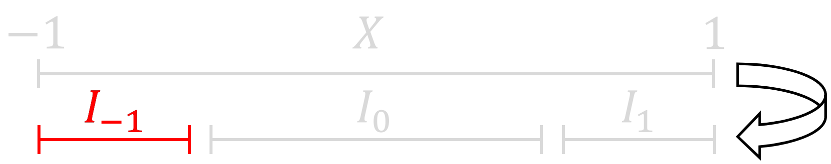


2) Divide the Input Range

- Assume bit-level operations are used as well
- To handle bit-level operations, divide X into intervals I_k,

so that, on each I_k, we can statically know
the result of each bit-level operation

- Example:



input x
y ← x $\times_f$ C
(C= 0x3ff71547652b82fe)
N ← ~~round(y)~~ -1

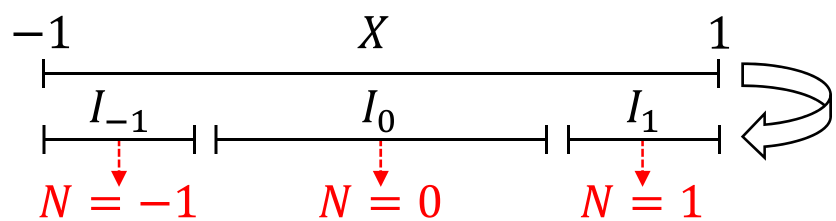
partial evaluation

input x
y ← x $\times_f$ C
(C= 0x3ff71547652b82fe)
N ← -1

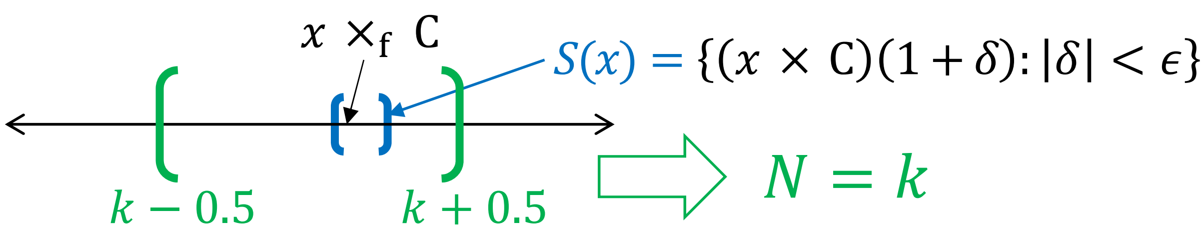
Only floating-point operations are left
→ Can compute $A_{\vec{\delta}}(x)$ on each I_k

2) Divide the Input Range

- How to find such intervals?
  - Use symbolic abstractions



- Example:
  - $N = \mathbf{round}(x \times_{\mathrm{f}} \mathrm{C})$
  - (symbolic abstraction of $x \times_{\mathrm{f}} \mathrm{C}$) = $(x \times \mathrm{C})(1 + \delta)$



  - Let I_k = largest interval contained in

$$\{x \in X : S(x) \subset (k - 0.5, k + 0.5)\}$$

  - Then N is evaluated to k for every input in I_k

3) Compute a Bound on Precision Loss

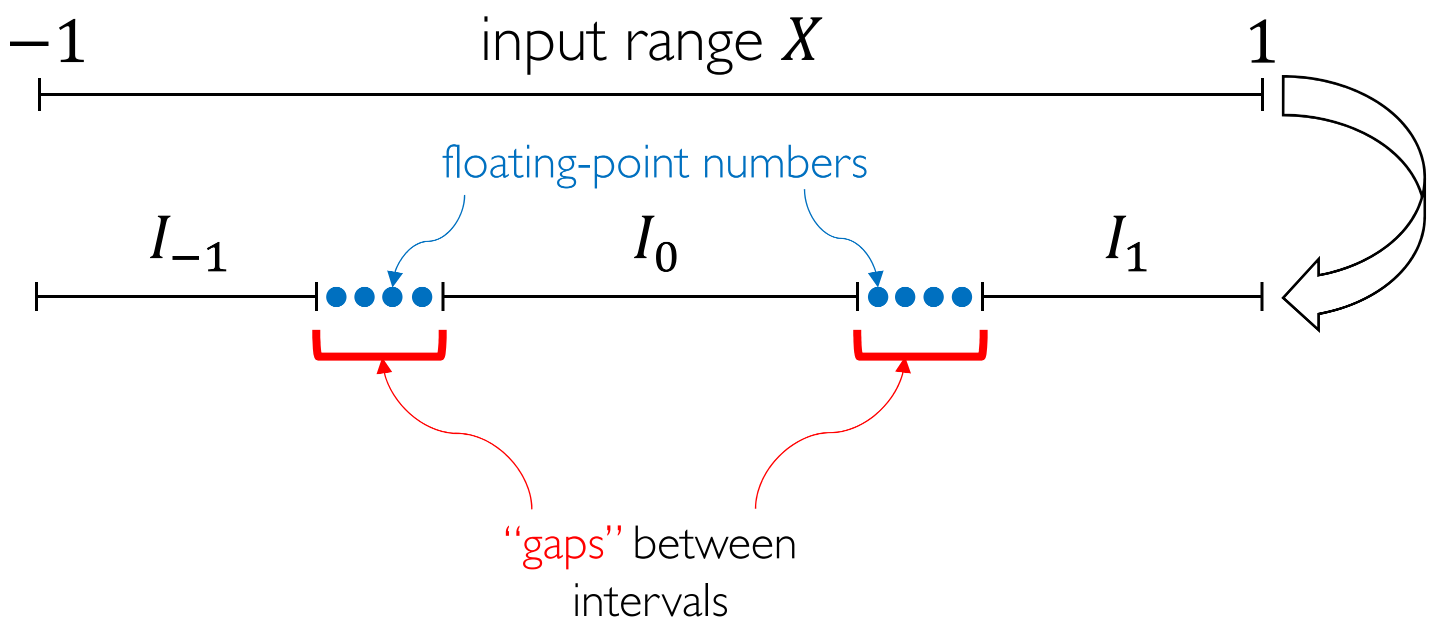
- Precision loss on each interval I_k
  - Let $A_{\vec{\delta}}(x)$ be a symbolic abstraction on I_k
  - Analytical optimization:

$$\max_{x \in I_k,\, |\delta_i| < \epsilon} \left| \frac{e^x - A_{\vec{\delta}}(x)}{e^x} \right|$$

  - Use Mathematica to solve optimization problems analytically

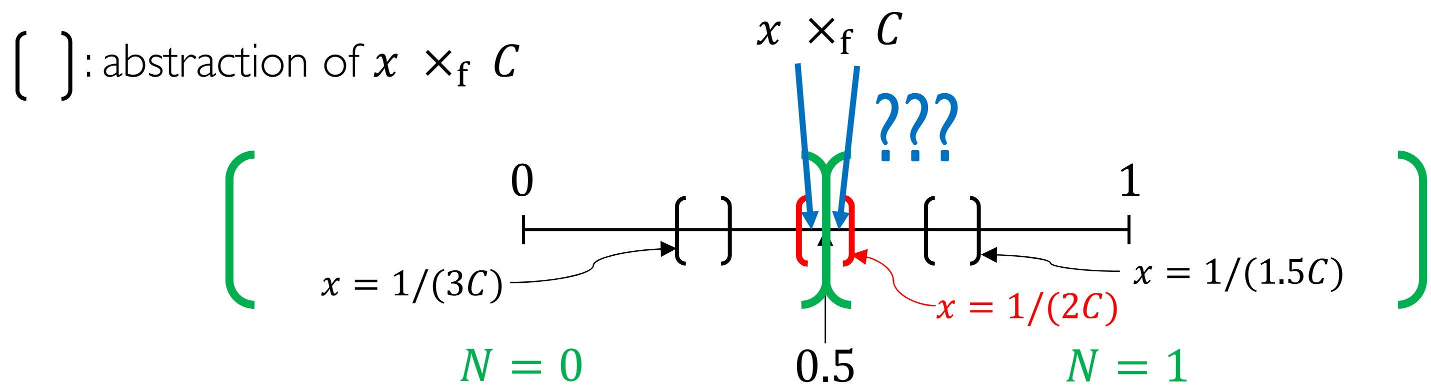
Are We Done?

- No. The constructed intervals do not cover X in general
  - Because we made sound approximations



Are We Done?

- Example: $N = \mathbf{round}(x \times_{\mathrm{f}} C)$



For $x = \frac{1}{2C}$, we can't statically know if N would be $\mathbf{0}$ or $\mathbf{1}$

- Let H = {floating-point numbers in the "gaps"}
  - We observe that $|H|$ is small in experiment

3) Compute a Bound on Precision Loss

- Precision loss on each interval I_k
  - Let $A_{\vec{\delta}}(x)$ be a symbolic abstraction on I_k
  - Analytical optimization:

$$\max_{x \in I_k,\, |\delta_i| < \epsilon} \left| \frac{e^x - A_{\vec{\delta}}(x)}{e^x} \right|$$

  - Use Mathematica to solve optimization problems analytically

take maximum

→ answer!

- Precision loss on H
  - For each $x \in H$, obtain $P(x)$ by executing the binary
  - Brute force:

$$\max_{x \in H} \left| \frac{e^x - P(x)}{e^x} \right|$$

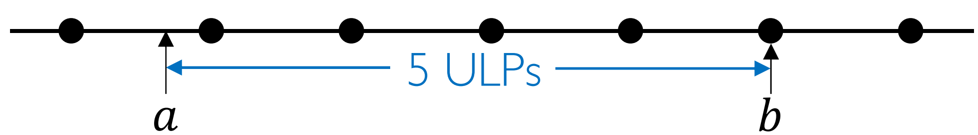
  - Use Mathematica to compute e^x and precision loss exactly

Case Studies

Settings

- Benchmarks
  - `exp`: from S3D (a combustion simulation engine)
  - `sin`, `log`: from Intel's <math.h>

- Measures of precision loss
  - Relative error: $\mathbf{RelErr}(a, b) = \left|\frac{a-b}{a}\right|$
  - ULP error:
    - Rounding errors of numeric libraries are typically measured by ULPs
    - $\mathbf{ULPErr}(a, b)$ = (# of floating-point numbers between a and b)
    - Example:



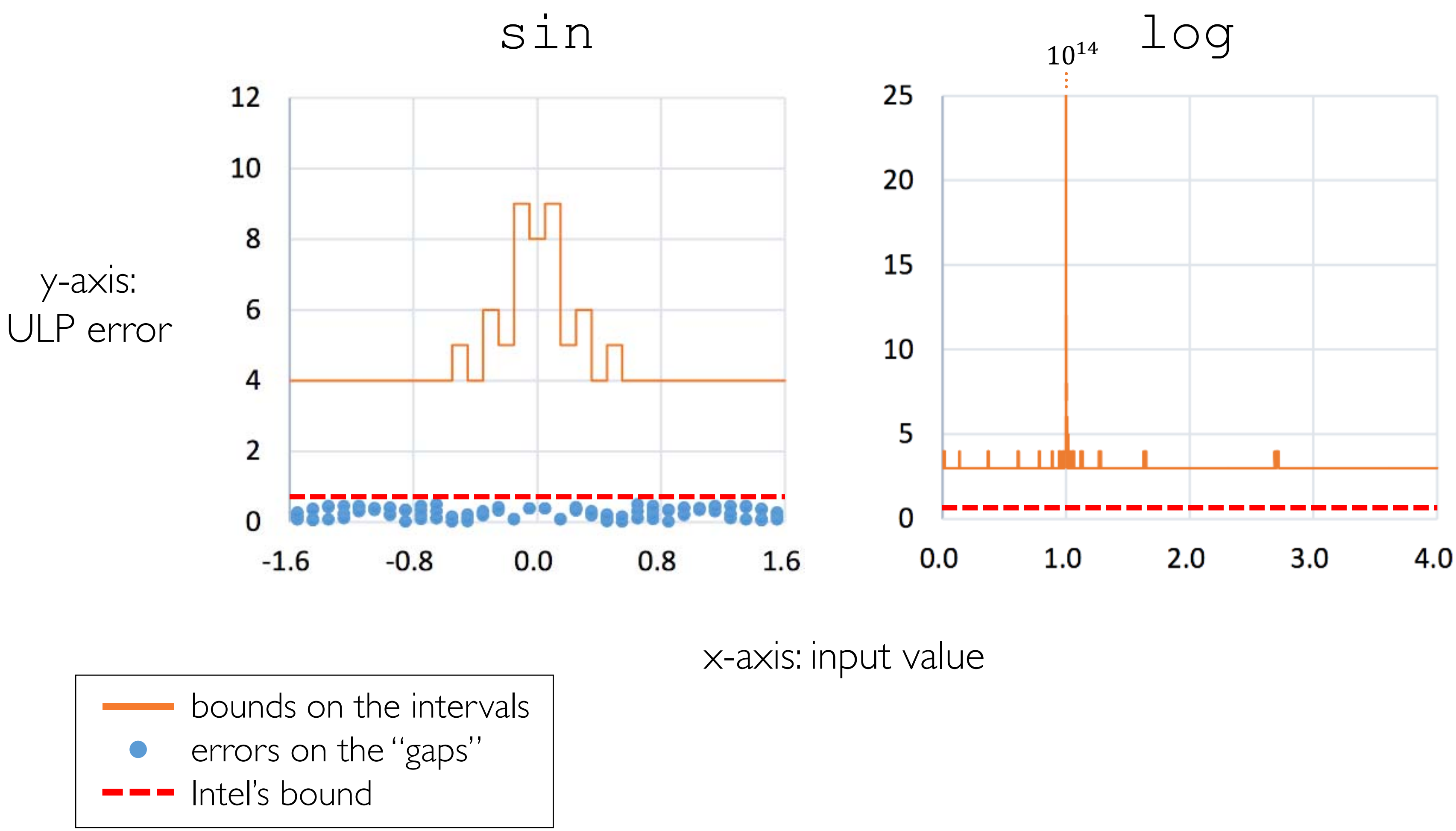
  - $\mathbf{ULPErr}(a, b) \leq 2 \cdot \mathbf{RelErr}(a, b)/\epsilon$

Results

	Interval	Bound on ULP error	# of intervals	# of δ's	Size of "gaps"
`exp`	$[-4, 4]$	14	13	29	36
`sin`	$\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	9	33	53	110
`log`	$(0,4) \setminus \left[\frac{4095}{4096}, 1\right)$	21	2^{21}	25	0
	$\left[\frac{4095}{4096}, 1\right)$	1×10^{14}	1	25	0

best illustrates
the power of our method

Results: sin, log



Limitations of Our Method

- May construct a large number of intervals
  - Example: `0x5fe6eb50c7b537a9 - (x >> 1)`
  - For this example, our method constructs 2^{63} intervals

- May produce loose error bounds
  - Example: 10^{14} ULPs for `log` on $\left[\frac{4095}{4096}, 1\right)$
  - Sometimes $(1+\epsilon)$ property provides an imprecise abstraction
  - Proving a precise error bound requires more sophisticated error analysis beyond $(1+\epsilon)$ property
  - Our recent result: 6 ULPs for for `log` on $(0,4)$

Summary

- First systematic method for verifying binaries that mix floating-point and bit-level operations
- Use abstraction, analytical optimization, and testing
- Directly applicable to highly optimized binaries of transcendental functions

Questions?