Verifying Bit-Manipulations of Floating-Point

Wonyeol Lee  Rahul Sharma  Alex Aiken
Stanford University

PLDI 2016
This Talk

- Example:

\[ e^x \]

(mathematical specification)
This Talk

• Example:

\[ e^x \]

mathematical specification

\[ ... \]

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

...
This Talk

• Example:

\[ e^x \]

mathematical specification

≠

floating-point implementation

\[
\begin{align*}
&... \\
vpslld & \, \text{\$20, } \%	ext{xmm3, } \%	ext{xmm3} \\
vpshufd & \, \text{\$114, } \%	ext{xmm3, } \%	ext{xmm3} \\
vmulpd & \, \text{C1, } \%	ext{xmm2, } \%	ext{xmm1} \\
vmulpd & \, \text{C2, } \%	ext{xmm2, } \%	ext{xmm2} \\
&... 
\end{align*}
\]
This Talk

- Example:

\[ e^x \]  

how different?

mathematical specification

\[
\begin{align*}
\text{vpslld} & \quad $20, \quad %xmm3, \quad %xmm3 \\
\text{vpshufd} & \quad $114, \quad %xmm3, \quad %xmm3 \\
\text{vmulpd} & \quad C1, \quad %xmm2, \quad %xmm1 \\
\text{vmulpd} & \quad C2, \quad %xmm2, \quad %xmm2 \\
\end{align*}
\]

floating-point implementation
This Talk

• Example:

\[ e^x \]

mathematical specification

how different?

• Goal: Bound the difference between spec and implementation
This Talk

- Example:
  \[ e^x \]
  mathematical specification
  \[ \text{how different?} \]

- Goal: Bound the **difference** between spec and implementation
- Key contribution: Verify binaries that mix floating-point and **bit-level operations**

```plaintext
... 
vpsslld $20, %xmm3, %xmm3 
vpshufd $114, %xmm3, %xmm3 
vmulpd C1, %xmm2, %xmm1 
vmulpd C2, %xmm2, %xmm2 
... 
```
This Talk

• Example:

\[ e^x \]

mathematical specification

how different?

• 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

...  

vpslld $20, %xmm3, %xmm3  
vpslfd $114, %xmm3, %xmm3  
vmulpd C1, %xmm2, %xmm1  
vmulpd C2, %xmm2, %xmm2  
...
Floating-Point Numbers

- Example:
  
  $\begin{align*}
  \text{Example:} & \quad 011111111111 \quad 1100\cdots00 \\
  & \quad = (-1)^1 \cdot 2^{1023-1023} \cdot 1.110\cdots00 \\
  & \quad = (-1)^1 \cdot 2^0 \cdot 1.110\cdots00 \\
  & \quad = (-1) \cdot 1.110\cdots00 \\
  & \quad = -1.110\cdots00 \\
  & \quad = -1.1100\cdots00 
  \end{align*}$
Floating-Point Numbers

- Example:
  
  \[
  \begin{align*}
  &\underbrace{1} 01111111111 1100\ldots00 \\
  &= (-1)^1 \cdot 2^{1023} - 1023 \cdot 1.110\ldots00_{(2)}
  \end{align*}
  \]

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

- Example:
  \[ 1.01111111111 \times 2^{1023} - 1023 \times 1.1100\cdots00(2) \]

- 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`)
Bit-Level Operations

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

Here $N = 10$

- Bit-shifting by $N$
- Converting from `int` to `double`
Bit-Level Operations

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

Here $N = 10$

- bit-shifting by $N$
- converting from `int` to `double`
- expensive

$2^N$ [int]

$2^N$ [double]
Bit-Level Operations

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

Here $N = 10$

- bit-shifting by $N$
- converting from `int` to `double`

 works only for $0 \leq N \leq 31$
Bit-Level Operations

- Example: Given \( N \) (in \texttt{int}), compute \( 2^N \) (in \texttt{double})

Here \( N = 10 \)

- Bit-shifting by \( N \)
- \( 2^N \) (in \texttt{int})

- Integer addition
- \( N + 1023 \) (in \texttt{int})

- Bit-shifting by 52
- [12 bits] \( 00 \ldots 0 \) (52 bits)

Works only for \( 0 \leq N \leq 31 \)

Expensive

Converting from \texttt{int} to \texttt{double}
Bit-Level Operations

• Example: Given \( N \) (in int), compute \( 2^N \) (in double)

here \( N = 10 \)

bit-shifting by \( N \)

works only for \( 0 \leq N \leq 31 \)

expensive

converting from int to double

integer addition

bit-shifting by 52

works only for \( 0 \leq N \leq 31 \)
Bit-Level Operations

- Example: Given \( N \) (in \texttt{int}), compute \( 2^N \) (in \texttt{double})

Here \( N = 10 \):

1. \( 1 \) [\texttt{int}]
2. \( 2^N \) [\texttt{int}]
3. \( 2^N \) [\texttt{double}]

This works only for \( 0 \leq N \leq 31 \)

Converting from \texttt{int} to \texttt{double} is expensive.

\( N + 1023 \) [\texttt{int}]

1. Integer addition
2. Bit-shifting by 52
3. \( 2^N \) [\texttt{double}]

This works for \( -1022 \leq N \leq 1023 \)
Bit-Level Operations

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

here $N = 10$

bit-shifting by $N$

expensive

converting from `int` to `double`

works only for $0 \leq N \leq 31$

$1$ [int]

$2^N$ [int]

$N$ [int]

$N + 1023$ [int]

integer addition

bit-shifting by 52

$2^N$ [double]

$2^N$ [double]

works for $-1022 \leq N \leq 1023$

• 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

• Goal: Find a small $\Theta > 0$ such that $f(x) - P(x) f(x) \leq \Theta$ for all $x \in X$.

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

$e^x$

mathematical specification

$f: \mathbb{R} \rightarrow \mathbb{R}$
Problem Statement

$e^x$

mathematical specification

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

binary $P$ that mixes floating-point and bit-level operations
Problem Statement

\[ e^x \]

mathematical specification

\( f : \mathbb{R} \rightarrow \mathbb{R} \)

input range \( X \subseteq \mathbb{R} \)

... 

\[
\begin{align*}
  \text{vpslld} & \quad 20, \quad %\text{xmm3}, \quad %\text{xmm3} \\
  \text{vpshufd} & \quad 114, \quad %\text{xmm3}, \quad %\text{xmm3} \\
  \text{vmulpd} & \quad \text{C1}, \quad %\text{xmm2}, \quad %\text{xmm1} \\
  \text{vmulpd} & \quad \text{C2}, \quad %\text{xmm2}, \quad %\text{xmm2}
\end{align*}
\]

binary \( P \) that mixes floating-point and bit-level operations
Problem Statement

- **Goal:** Find a small $\Theta > 0$ such that
  \[
  \left| \frac{f(x) - P(x)}{f(x)} \right| \leq \Theta \quad \text{for all } x \in X
  \]
  - i.e., prove a bound on the maximum precision loss

$e^x$

mathematical specification

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

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

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

binary $P$ that mixes floating-point and bit-level operations
Possible Alternatives

- Exhaustive testing
  - feasible for 32-bit float: \(\sim 30\) seconds (with 1 core for \(\sin f\))
  - infeasible for 64-bit double: \(> 4000\) years (\(= 30\) seconds \(\times 2^{32}\))

\[\because\quad (\text{# of doubles between } -1 \text{ and } 1) = \frac{1}{2} (\text{# of all doubles})\]
Possible Alternatives

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

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

• Machine-checkable proofs
  • Harrison used HOL Light to prove Intel’s transcendental functions are very accurate [FMCAD’00]
Possible Alternatives

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

• 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], RO SA [POPL’14], FPTaylor [FM’15]

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

- If only floating-point operations are used, various automatic techniques can be applied
  - e.g., Astree [PLDI’03], Fluctuat [FMICS’09], RO SA [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
\[ e^x \text{ Explained} \]

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>vmovddup</td>
<td>%xmm0, %xmm0</td>
</tr>
<tr>
<td>2</td>
<td>vmulpd</td>
<td>L2E, %xmm0, %xmm2</td>
</tr>
<tr>
<td>3</td>
<td>vroundpd</td>
<td>$0, %xmm2, %xmm2</td>
</tr>
<tr>
<td>4</td>
<td>vcvtpd2dqx</td>
<td>%xmm2, %xmm3</td>
</tr>
<tr>
<td>5</td>
<td>vpaddb</td>
<td>B, %xmm3, %xmm3</td>
</tr>
<tr>
<td>6</td>
<td>vpslld</td>
<td>$20, %xmm3, %xmm3</td>
</tr>
<tr>
<td>7</td>
<td>vpshufd</td>
<td>$114, %xmm3, %xmm3</td>
</tr>
<tr>
<td>8</td>
<td>vmulpd</td>
<td>C1, %xmm2, %xmm1</td>
</tr>
<tr>
<td>9</td>
<td>vmulpd</td>
<td>C2, %xmm2, %xmm2</td>
</tr>
<tr>
<td>10</td>
<td>vaddpd</td>
<td>%xmm1, %xmm0, %xmm1</td>
</tr>
<tr>
<td>11</td>
<td>vaddpd</td>
<td>%xmm2, %xmm1, %xmm1</td>
</tr>
<tr>
<td>12</td>
<td>vmovapd</td>
<td>T1, %xmm0</td>
</tr>
<tr>
<td>13</td>
<td>vmulpd</td>
<td>T12, %xmm1, %xmm2</td>
</tr>
<tr>
<td>14</td>
<td>vaddpd</td>
<td>T11, %xmm2, %xmm2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>vaddpd</td>
<td>%xmm0, %xmm1, %xmm0</td>
</tr>
<tr>
<td>37</td>
<td>vmulpd</td>
<td>%xmm3, %xmm0, %xmm0</td>
</tr>
<tr>
<td>38</td>
<td>retq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction</td>
<td>Operands</td>
</tr>
<tr>
<td>---</td>
<td>-------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>movddup</td>
<td>%xmm0, %xmm0</td>
</tr>
<tr>
<td>2</td>
<td>vmulpd</td>
<td>L2E, %xmm0, %xmm2</td>
</tr>
<tr>
<td>3</td>
<td>vroundpd</td>
<td>$0, %xmm2, %xmm2</td>
</tr>
<tr>
<td>4</td>
<td>vcvtpd2dqx</td>
<td>%xmm2, %xmm3</td>
</tr>
<tr>
<td>5</td>
<td>vpaddd</td>
<td>B, %xmm3, %xmm3</td>
</tr>
<tr>
<td>6</td>
<td>vpslld</td>
<td>$20, %xmm3, %xmm3</td>
</tr>
<tr>
<td>7</td>
<td>vpshufd</td>
<td>$114, %xmm3, %xmm3</td>
</tr>
<tr>
<td>8</td>
<td>vmulpd</td>
<td>C1, %xmm2, %xmm1</td>
</tr>
<tr>
<td>9</td>
<td>vmulpd</td>
<td>C2, %xmm2, %xmm2</td>
</tr>
<tr>
<td>10</td>
<td>vaddpd</td>
<td>%xmm1, %xmm0, %xmm1</td>
</tr>
<tr>
<td>11</td>
<td>vaddpd</td>
<td>%xmm2, %xmm1, %xmm1</td>
</tr>
<tr>
<td>12</td>
<td>movapd</td>
<td>T1, %xmm0</td>
</tr>
<tr>
<td>13</td>
<td>vmulpd</td>
<td>T12, %xmm1, %xmm2</td>
</tr>
<tr>
<td>14</td>
<td>vaddpd</td>
<td>T11, %xmm2, %xmm2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>vaddpd</td>
<td>%xmm0, %xmm1, %xmm0</td>
</tr>
<tr>
<td>37</td>
<td>vmulpd</td>
<td>%xmm3, %xmm0, %xmm0</td>
</tr>
<tr>
<td>38</td>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>

\[ N = \text{round}(x \cdot \log_2 e) \]
\( e^x \) Explained

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>vmovdudp</code></td>
<td><code>%xmm0</code>, <code>%xmm0</code></td>
</tr>
<tr>
<td>2</td>
<td><code>vmulpd</code></td>
<td><code>L2E</code>, <code>%xmm0</code>, <code>%xmm2</code></td>
</tr>
<tr>
<td>3</td>
<td><code>vroundpd</code></td>
<td><code>$0$, </code>%xmm2<code>, </code>%xmm2`</td>
</tr>
<tr>
<td>4</td>
<td><code>vcvtpd2dqx</code></td>
<td><code>%xmm2</code>, <code>%xmm3</code></td>
</tr>
<tr>
<td>5</td>
<td><code>vpadddd</code></td>
<td><code>B</code>, <code>%xmm3</code>, <code>%xmm3</code></td>
</tr>
<tr>
<td>6</td>
<td><code>vpslld</code></td>
<td><code>$20$, </code>%xmm3<code>, </code>%xmm3`</td>
</tr>
<tr>
<td>7</td>
<td><code>vpshufd</code></td>
<td><code>$114$, </code>%xmm3<code>, </code>%xmm3`</td>
</tr>
<tr>
<td>8</td>
<td><code>vmulpd</code></td>
<td><code>C1</code>, <code>%xmm2</code>, <code>%xmm1</code></td>
</tr>
<tr>
<td>9</td>
<td><code>vmulpd</code></td>
<td><code>C2</code>, <code>%xmm2</code>, <code>%xmm2</code></td>
</tr>
<tr>
<td>10</td>
<td><code>vaddpd</code></td>
<td><code>%xmm1</code>, <code>%xmm0</code>, <code>%xmm1</code></td>
</tr>
<tr>
<td>11</td>
<td><code>vaddpd</code></td>
<td><code>%xmm2</code>, <code>%xmm1</code>, <code>%xmm1</code></td>
</tr>
<tr>
<td>12</td>
<td><code>vmovapd</code></td>
<td><code>T1</code>, <code>%xmm0</code></td>
</tr>
<tr>
<td>13</td>
<td><code>vmulpd</code></td>
<td><code>T12</code>, <code>%xmm1</code>, <code>%xmm2</code></td>
</tr>
<tr>
<td>14</td>
<td><code>vaddpd</code></td>
<td><code>T11</code>, <code>%xmm2</code>, <code>%xmm2</code></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td><code>vaddpd</code></td>
<td><code>%xmm0</code>, <code>%xmm1</code>, <code>%xmm0</code></td>
</tr>
<tr>
<td>37</td>
<td><code>vmulpd</code></td>
<td><code>%xmm3</code>, <code>%xmm0</code>, <code>%xmm0</code></td>
</tr>
<tr>
<td>38</td>
<td><code>retq</code></td>
<td></td>
</tr>
</tbody>
</table>
$e^x$ Explained

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

\[ 2^N \]

\[ r = x - N \cdot \ln 2 \]

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

\[ e^x = e^{N \cdot \ln 2} \cdot e^r \approx 2^N \cdot e^r \]
\[ e^x \text{ Explained} \]

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Source Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>vmovddup</td>
<td>%xmm0, %xmm0</td>
</tr>
<tr>
<td>2</td>
<td>vmulpd</td>
<td>L2E, %xmm0, %xmm2</td>
</tr>
<tr>
<td>3</td>
<td>vroundpd</td>
<td>$0, %xmm2, %xmm2</td>
</tr>
<tr>
<td>4</td>
<td>vcvtfd2dq</td>
<td>%xmm2, %xmm3</td>
</tr>
<tr>
<td>5</td>
<td>vpadd</td>
<td>B, %xmm3, %xmm3</td>
</tr>
<tr>
<td>6</td>
<td>vpslld</td>
<td>$20, %xmm3, %xmm3</td>
</tr>
<tr>
<td>7</td>
<td>vpshufd</td>
<td>$114, %xmm3, %xmm3</td>
</tr>
<tr>
<td>8</td>
<td>vmulpd</td>
<td>C1, %xmm2, %xmm1</td>
</tr>
<tr>
<td>9</td>
<td>vmulpd</td>
<td>C2, %xmm2, %xmm2</td>
</tr>
<tr>
<td>10</td>
<td>vaddpd</td>
<td>%xmm1, %xmm0, %xmm1</td>
</tr>
<tr>
<td>11</td>
<td>vaddpd</td>
<td>%xmm2, %xmm1, %xmm1</td>
</tr>
<tr>
<td>12</td>
<td>vmovapd</td>
<td>T1, %xmm0</td>
</tr>
<tr>
<td>13</td>
<td>vmulpd</td>
<td>T12, %xmm1, %xmm2</td>
</tr>
<tr>
<td>14</td>
<td>vaddpd</td>
<td>T11, %xmm2, %xmm2</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>36</td>
<td>vaddpd</td>
<td>%xmm0, %xmm1, %xmm0</td>
</tr>
<tr>
<td>37</td>
<td>vmulpd</td>
<td>%xmm3, %xmm0, %xmm0</td>
</tr>
<tr>
<td>38</td>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ 2^N \]

\[ r = x - N \cdot \ln 2 \]

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

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

**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) Abstract Floating-Point Operations

• Assume only floating-point operations are used
• \((1 + \epsilon)\) property
  • A standard way to model rounding errors
1) Abstract Floating-Point Operations

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

\[ x \otimes_f y \in \{(x \otimes y)(1 + \delta) : |\delta| < \varepsilon\} \]
1) Abstract Floating-Point Operations

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

\[
x \otimes_f y \in \{(x \otimes y)(1 + \delta) : |\delta| < \epsilon\}
\]
1) Abstract Floating-Point Operations

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

\[ x \otimes_f y \in \{(x \otimes y)(1 + \delta) : |\delta| < \epsilon\} \]
1) Abstract Floating-Point Operations

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

\[ x \otimes_f y \in \{(x \otimes y)(1 + \delta) : |\delta| < \epsilon\} \]
1) Abstract Floating-Point Operations

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

\[ x \otimes_f y \in \{(x \otimes y)(1 + \delta) : |\delta| < \varepsilon\} \]

- For 64-bit doubles, \(\varepsilon = 2^{-53}\)
1) Abstract Floating-Point Operations

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

\[ x \otimes_f y \in \{(x \otimes y)(1 + \delta) : |\delta| < \epsilon\} \]

- 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 $A_\delta(x)$ of a program $P$.
1) Abstract Floating-Point Operations

• Compute a symbolic abstraction $\mathcal{A}_{\delta}(x)$ of a program $P$
  • Example:

  $$P(x) = ((2 \times_f x) +_f 3)$$
1) Abstract Floating-Point Operations

- Compute a symbolic abstraction $A_{\delta}(x)$ of a program $P$
  - Example:
    $$A_{\delta}(x) = ((2 \times_f x) +_f 3)$$
1) Abstract Floating-Point Operations

- Compute a symbolic abstraction $A_{\delta}(x)$ of a program $P$
  - Example:
    
    $A_{\delta}(x) = ((2 \times x) + 3)$
1) Abstract Floating-Point Operations

- Compute a **symbolic abstraction** $A_\delta(x)$ of a program $P$
  - Example:
    $$A_\delta(x) = ((2 \times x)(1 + \delta_1) + 3)(1 + \delta_2)$$
1) Abstract Floating-Point Operations

- Compute a symbolic abstraction $A_{\delta}(x)$ of a program $P$
  - Example:
    $$A_{\delta}(x) = ((2 \times x)(1 + \delta_1) + 3)(1 + \delta_2)$$

- From $(1 + \epsilon)$ property, $A_{\delta}(x)$ satisfies
  $$P(x) \in \{A_{\delta}(x) : |\delta_i| < \epsilon\} \text{ for all } x$$
1) Abstract Floating-Point Operations

- Compute a symbolic abstraction $A_{\delta}(x)$ of a program $P$
  - Example:
    $$A_{\delta}(x) = ((2 \times x)(1 + \delta_1) + 3)(1 + \delta_2)$$

- From $(1 + \epsilon)$ property, $A_{\delta}(x)$ satisfies
  $$P(x) \in \{A_{\delta}(x) : |\delta_i| < \epsilon\} \text{ for all } x$$

  - Example:
    $$P(x) = ((2 \times_f x) +_f 3)$$
1) Abstract Floating-Point Operations

- Compute a **symbolic abstraction** $A_\delta(x)$ of a program $P$
  - Example:
    $$A_\delta(x) = ((2 \times x)(1 + \delta_1) + 3)(1 + \delta_2)$$

- From $(1 + \epsilon)$ property, $A_\delta(x)$ satisfies
  $$P(x) \in \{A_\delta(x) : |\delta_i| < \epsilon\} \text{ for all } x$$
  - Example:
    $$P(x) \subseteq \{(2 \times x)(1 + \delta_1) + 3)(1 + \delta_2) : |\delta_1|, |\delta_2| < \epsilon\}$$
1) Abstract Floating-Point Operations

- Compute a symbolic abstraction $A_{\delta}(x)$ of a program $P$
  - Example:
    $A_{\delta}(x) = ((2 \times x)(1 + \delta_1) + 3)(1 + \delta_2)$

- From $(1 + \epsilon)$ property, $A_{\delta}(x)$ satisfies
  $P(x) \in \{A_{\delta}(x) : |\delta_i| < \epsilon\}$ for all $x$

  - Example:
    $P(x) \in \{((2 \times x)(1 + \delta_1) + 3)(1 + \delta_2) : |\delta_1|, |\delta_2| < \epsilon\}$
Our Method: Overview

\[ P(x) \]

\[ X \]

\[ -1 \quad 1 \]

\[
\begin{align*}
\text{vpslld} & \quad \$20, & \ldots \\
\text{vpshufd} & \quad \$114, & \ldots \\
\text{vmulpd} & \quad \text{C1,} & \ldots \\
\text{vmulpd} & \quad \text{C2,} & \ldots \\
\ldots
\end{align*}
\]
Our Method: Overview

\[ P(x) \]

\[ X \]

\[ -1 \quad 1 \]

\[
\begin{align*}
\text{\texttt{vpslld}} & \quad \$20, \quad \ldots \\
\text{\texttt{vpshufd}} & \quad $114, \quad \ldots \\
\text{\texttt{vmulpd}} & \quad C1, \quad \ldots \\
\text{\texttt{vmulpd}} & \quad C2, \quad \ldots \\
\ldots & 
\end{align*}
\]
Our Method: Overview

\[ P(x) \]

\[ \cdots \]
\[
\begin{array}{c}
\text{vpslld} & \$20, \ \cdots \\
\text{vpshufd} & \$114, \ \cdots \\
\text{vmulpd} & C1, \ \cdots \\
\text{vmulpd} & C2, \ \cdots \\
\end{array}
\]
Our Method: Overview

\[ P(x) \]

hard to find

\[ -1 \quad 1 \]

\[ X \]

... 

\texttt{vpslld} $20, \ldots$

\texttt{vpshufd} $114, \ldots$

\texttt{vmulpd} C1, \ldots

\texttt{vmulpd} C2, \ldots

...
Our Method: Overview

\[ P(x) \]

hard to find

\[ -1 \quad X \quad 1 \]

not “smooth”

abstract using “smooth” functions

\[ \begin{align*}
\text{vpslld} & \quad $20, \quad \ldots \\
\text{vpshufd} & \quad $114, \quad \ldots \\
\text{vmulpd} & \quad C1, \quad \ldots \\
\text{vmulpd} & \quad C2, \quad \ldots
\end{align*} \]
Our Method: Overview

$P(x)$

hard to find

not “smooth”

abstract using “smooth” functions

$vpslld$ $20,$ ...
$vpslld$ $114,$ ...
$vmulpd$ $C1,$ ...
$vmulpd$ $C2,$ ...

abstract using “smooth” functions

$I_1$ $I_2$ ... $I_n$
Our Method: Overview

$P(x)$

hard to find

not “smooth”

abstract using “smooth” functions

$-1 \quad 1$

\[ x \]

\[
\begin{array}{l}
\ldots \\
vpslld \; \$20, \; \ldots \\
vpshufd \; \$114, \; \ldots \\
vmulpd \; C1, \; \ldots \\
vmulpd \; C2, \; \ldots \\
\ldots 
\end{array}
\]

\[
\begin{array}{l}
\ldots \\
vpslld \; \$20, \; \ldots \\
vpshufd \; \$114, \; \ldots \\
vmulpd \; C1, \; \ldots \\
vmulpd \; C2, \; \ldots \\
\ldots 
\end{array}
\]

\[
\begin{array}{l}
\ldots \\
vpslld \; \$20, \; \ldots \\
vpshufd \; \$114, \; \ldots \\
vmulpd \; C1, \; \ldots \\
vmulpd \; C2, \; \ldots \\
\ldots 
\end{array}
\]

\[
\begin{array}{l}
\ldots \\
vpslld \; \$20, \; \ldots \\
vpshufd \; \$114, \; \ldots \\
vmulpd \; C1, \; \ldots \\
vmulpd \; C2, \; \ldots \\
\ldots 
\end{array}
\]
Our Method: Overview

\[ P(x) \]

hard to find

not “smooth”

abstract using “smooth” functions

partial evaluation of bit-level operations

\[ -1 \quad X \quad 1 \]

\[
\begin{align*}
&vpslld \quad \text{\$20, \ldots} \\
vphufd \quad \text{\$114, \ldots} \\
vmulpd \quad \text{C1, \ldots} \\
vmulpd \quad \text{C2, \ldots}
\end{align*}
\]

\[ 1 \quad 3 \quad 2n + 1 \]

\[ n \]

\[ l_1 \quad l_2 \quad \ldots \quad l_n \]

hard to find
Our Method: Overview

\[ P(x) \]

Hard to find

-1 \[ \rightarrow \] 1

\[ X \]

Not "smooth"

Abstract using "smooth" functions

Only floating-point operations

Partial evaluation of bit-level operations

\[ n \]

\[ 2n + 1 \]
Our Method: Overview

- $P(x)$ hard to find
- $A_1,\delta(x)$, $A_2,\delta(x)$, ..., $A_n,\delta(x)$
- $-1 \leq x \leq 1$
- not "smooth"
- abstract using "smooth" functions
- only floating-point operations

abstract using "smooth" functions

partial evaluation of bit-level operations

$\cdots$
$vpslld \ 20, \ \cdots$
$vpslld \ 114, \ \cdots$
$vmulpd \ C1, \ \cdots$
$vmulpd \ C2, \ \cdots$
$\cdots$

$1$
$3$
$n$

$2n + 1$
Our Method: Overview

\[ A_{1, \bar{\delta}}(x) \rightarrow A_{2, \bar{\delta}}(x) \rightarrow A_{n, \bar{\delta}}(x) \]

\[ I_1, I_2, \ldots, I_n \]

- \text{vpslld} $20, \ldots$
- \text{vpshufd} $114, \ldots$
- \text{vmulpd} C1, \ldots
- \text{vmulpd} C2, \ldots

Partial evaluation of bit-level operations

1. \text{vpslld} $20, \ldots$
2. \text{vpshufd} $114, \ldots$
3. \text{vmulpd} C1, \ldots
4. \text{vmulpd} C2, \ldots

1. \text{vpslld} $20, \ldots$
2. \text{vpshufd} $114, \ldots$
3. \text{vmulpd} C1, \ldots
4. \text{vmulpd} C2, \ldots

\( n \)

\( 2n + 1 \)
Our Method: Overview

\[ f(x) - A_{1,\delta}(x) \]
\[ f(x) - A_{n,\delta}(x) \]

\[ \frac{f(x) - A_{1,\delta}(x)}{f(x)} \]
\[ \frac{f(x) - A_{n,\delta}(x)}{f(x)} \]

partial evaluation of bit-level operations
Our Method: Overview

\[ A_{1,\bar{\delta}}(x) \quad A_{2,\bar{\delta}}(x) \quad \ldots \quad A_{n,\bar{\delta}}(x) \]

\[ f(x) - A_{1,\delta}(x) \quad f(x) - A_{n,\delta}(x) \]

\[ \text{solve optimization problems} \]

\[
\max \left| \frac{f(x) - A_{1,\delta}(x)}{f(x)} \right| \quad \max \left| \frac{f(x) - A_{n,\delta}(x)}{f(x)} \right|
\]

\[ l_1 \quad l_2 \quad \ldots \quad l_n \]

\[
\text{...}
\text{vpadd} \quad \text{vpadd} \quad \text{vpadd} \quad \text{vpadd} \quad \ldots
\]

\[
\text{vpslld} \quad 20, \quad \text{vpshufd} \quad 114, \quad \text{vmulpd} \quad C_1, \quad \text{vmulpd} \quad C_2, \quad \ldots
\]

\[
\text{partial evaluation of bit-level operations}
\]

\[ \max 1 \quad \max 3 \quad \max n \quad \max 2n + 1 \]
Our Method: Overview

\[ A_{1,\delta}(x) \quad A_{2,\delta}(x) \quad \ldots \quad A_{n,\delta}(x) \]

\[ f(x) - A_1(x) \quad \delta x \quad f(x) - A_n(x) \]

\[ \max \left| \frac{f(x) - A_{1,\delta}(x)}{f(x)} \right| \]

\[ \max \left| \frac{f(x) - A_{n,\delta}(x)}{f(x)} \right| \]

solve optimization problems

\[ I_1 \quad I_2 \quad \ldots \quad I_n \]

\[ \text{partial evaluation of bit-level operations} \]

\[ \text{answer!} \]
2) Divide the Input Range

• Assume bit-level operations are used as well
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
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 × f C
   (C=0x3ff71547652b82fe)
N ← round(y)
z ← int(N) + i 0x3ff
w ← z << 52
...
```
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:

```python
input x
y ← x ×_f C  
   (C=0x3ff71547652b82fe)
N ← round(y)
z ← int(N) +i 0x3ff
w ← z << 52
...
```
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 ×_f C
    (C=0x3ff71547652b82fe)
N ← round(y)
z ← int(N) +_i 0x3ff
w ← z << 52
...
```
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 ← \text{round}(y)  
z ← \text{int}(N) + 0x3ff
w ← z \ll 52
...  
```
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 ×_f C  
  (C=0x3ff71547652b82fe)
N ← round(y)  
  -1
z ← int(N) +i 0x3ff
w ← z << 52
...
```

```
input x
y ← x ×_f C  
  (C=0x3ff71547652b82fe)
N ← -1
z ← 1022
w ← 0.5
...
```
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:

```
<table>
<thead>
<tr>
<th>input x</th>
<th>y ← x \times_f C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(C=0x3ff71547652b82fe)</td>
</tr>
</tbody>
</table>
N ← round(y)  | 1                  |
```

Only floating-point operations are left
→ Can compute $A_{\delta}(x)$ on each $I_k$
2) Divide the Input Range

• How to find such intervals?

\[ N = \text{round} \times f_C \]

\[ (\text{symbolic abstraction of } x \times f_C) = x \times C_1 + \delta \]

• Let \( I_k = \text{largest interval contained in } x \in X : S_x \subset k - 0.5, k + 0.5 \).

• Then \( N \) is evaluated to \( k \) for every input in \( I_k \).
2) Divide the Input Range

- How to find such intervals?

\[ N = \text{round } x \times f_C \]

\[ (\text{symbolic abstraction of } x \times f_C) = x \times C_1 + \delta \]

Let \( I_k = \text{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 \( I_{k-1} I_0 I_1 \).
2) Divide the Input Range

• How to find such intervals?

\[ N = \text{round } x \times f \]

\[ (\text{symbolic abstraction of } x \times f) = x \times C_1 + \delta \]

Let \( I_k = \text{largest interval contained in } x \in X : S_x \subset k - 0.5, k + 0.5 \).

Then \( N \) is evaluated to \( k \) for every input in \( I_k \).
2) Divide the Input Range

• How to find such intervals?
  • Use symbolic abstractions

\[
\text{Example: } N = \text{round} \ x \times f_C \quad \text{(symbolic abstraction of } x \times f_C) = x \times c_1 + \delta
\]

Let \( I_k = \text{largest interval contained in } x \in X : S_x \subset k - 0.5, k + 0.5. \)

Then \( N \) is evaluated to \( k \) for every input in \( I_k \).
2) Divide the Input Range

• How to find such intervals?
  • Use symbolic abstractions

• Example:
  • \( N = \text{round}(x \times_f C) \)

\[\begin{align*}
\mathcal{N} &= -1 \\
\mathcal{I}_-1 &\quad \mathcal{I}_0 \quad \mathcal{I}_1 \\
N &= -1 \quad N = 0 \quad N = 1
\end{align*}\]
2) Divide the Input Range

• How to find such intervals?
  • Use symbolic abstractions

• Example:
  • $N = \text{round}(x \times_f C)$
  • (symbolic abstraction of $x \times_f C) = (x \times C)(1 + \delta)$
2) Divide the Input Range

• How to find such intervals?
  • Use symbolic abstractions

• Example:
  • $N = \text{round}(x \times_f C)$
  • (symbolic abstraction of $x \times_f C) = (x \times C)(1 + \delta)$
2) Divide the Input Range

- How to find such intervals?
  - Use symbolic abstractions

- Example:
  - \( N = \text{round}(x \times_f C) \)
  - (symbolic abstraction of \( x \times_f C \)) = \((x \times C)(1 + \delta)\)

\[
\begin{align*}
N &= -1 \\
N &= 0 \\
N &= 1
\end{align*}
\]
2) Divide the Input Range

• How to find such intervals?
  • Use symbolic abstractions

• Example:
  • \( N = \text{round}(x \times f \ C) \)
  • (symbolic abstraction of \( x \times f \ C \)) = \( (x \times C)(1 + \delta) \)

\[
\begin{align*}
N &= -1 \\
N &= 0 \\
N &= 1
\end{align*}
\]
2) Divide the Input Range

- How to find such intervals?
  - Use symbolic abstractions

- Example:
  - \( N = \text{round}(x \times_f C) \)
  - (symbolic abstraction of \( x \times_f C \)) = \((x \times C)(1 + \delta)\)

- Let \( I_k = \) largest interval contained in
  \[ \{x \in X : S(x) \subset (k - 0.5, k + 0.5)\} \]
2) Divide the Input Range

• How to find such intervals?
  • Use symbolic abstractions

• Example:
  • \( N = \text{round}(x \times_f C) \)
  • (symbolic abstraction of \( x \times_f C \)) = \((x \times C)(1 + \delta)\)

\[ S(x) = \{(x \times C)(1 + \delta): |\delta| < \epsilon\} \]

• 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 $A_{\delta}(x)$ be a symbolic abstraction on $I_k$
3) Compute a Bound on Precision Loss

• Precision loss on each interval $I_k$
  • Let $A_\delta(x)$ be a symbolic abstraction on $I_k$
  • Analytical optimization:
    \[
    \max_{x \in I_k, |\delta_i| < \epsilon} \left| \frac{e^x - A_\delta(x)}{e^x} \right|
    \]
  • Use Mathematica to solve optimization problems analytically
Are We Done?

- No. The constructed intervals do not cover $X$ in general.
Are We Done?

• No. The constructed intervals do not cover $X$ in general.
Are We Done?

- No. The constructed intervals do not cover $X$ in general
Are We Done?

No. The constructed intervals do not cover $X$ in general

- Because we made sound approximations
Are We Done?

- Example: \( N = \text{round}(x \times_f C) \)

\[ \{ \} : \text{abstraction of } x \times_f C \]

0 \[ \longrightarrow \] 0.5 \[ \longrightarrow \] 1
Are We Done?

- Example: $N = \text{round}(x \times_f C)$

\[
\{ \}: \text{abstraction of } x \times_f C
\]

\[
x = \frac{1}{3C}
\]

\[
x = \frac{1}{1.5C}
\]

\[
x = \frac{1}{0.5C}
\]
Are We Done?

- Example: \( N = \text{round}(x \times_f C) \)

\[
\begin{align*}
\{ \} & : \text{abstraction of } x \times_f C \\
\begin{cases}
0 & \text{if } x = 1/(3C) \\
0.5 & \text{if } x = 1/(1.5C) \\
1 & \text{otherwise}
\end{cases}
\end{align*}
\]
Are We Done?

- Example: $N = \text{round}(x \times_f C)$

\[
\begin{aligned}
&\left\{ \right. \text{abstraction of } x \times_f C \\
&x = 1/(3C) \quad 0 \quad \left\{ \right. \\
&x = 1/(2C) \quad 0.5 \quad N = 0 \\
&x = 1/(1.5C) \quad 1 \quad N = 1
\end{aligned}
\]
Are We Done?

- Example: \( N = \text{round}(x \times_f C) \)

\[ \{ \}: \text{abstraction of } x \times_f C \]

\[
\begin{align*}
N &= 0 \\
0 &\quad \{ \} \quad 0.5 \quad \{ \} \quad 1 \\
x &= 1/(3C) & \quad x = 1/(2C) & \quad x = 1/(1.5C)
\end{align*}
\]
Are We Done?

- Example: $N = \text{round}(x \times_f C)$

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

• Example: \( N = \text{round}(x \times_f C) \)

\[
\begin{cases}
0 & \text{if } N = 0 \\
0.5 & \text{if } x = 1/(3C) \\
1 & \text{if } x = 1/(2C) \\
\text{else} & \text{if } x = 1/(1.5C) \\
\end{cases}
\]

For \( x = \frac{1}{2C} \), we can’t statically know if \( N \) would be 0 or 1

• Let \( H = \{\text{floating-point numbers in the “gaps”}\} \)
  • We observe that \(|H|\) is small in experiment
3) Compute a Bound on Precision Loss

- Precision loss on each interval $I_k$
  - Let $A_\delta(x)$ be a symbolic abstraction on $I_k$
  - Analytical optimization:
    \[
    \max_{x \in I_k, \left|\delta_i\right| < \epsilon} \left| \frac{e^x - A_\delta(x)}{e^x} \right|
    \]
  - Use Mathematica to solve optimization problems analytically

- 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
3) Compute a Bound on Precision Loss

- Precision loss on each interval $I_k$
  - Let $A_\delta(x)$ be a symbolic abstraction on $I_k$
  - Analytical optimization:
    $$\max_{x \in I_k, |\delta_i| < \epsilon} \left| \frac{e^x - A_\delta(x)}{e^x} \right|$$
  - Use Mathematica to solve optimization problems analytically

- 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: \( \text{RelErr}(a, b) = \left| \frac{a-b}{a} \right| \)
  • ULP error:
    • Rounding errors of numeric libraries are typically measured by ULPs
• **Benchmarks**
  - exp: from S3D (a combustion simulation engine)
  - sin, log: from Intel’s `<math.h>`

• **Measures of precision loss**
  - Relative error: \( \text{RelErr}(a, b) = \left| \frac{a-b}{a} \right| \)
  - ULP error:
    - Rounding errors of numeric libraries are typically measured by ULPs
    - \( \text{ULPErr}(a, b) = (\text{# of floating-point numbers between } a \text{ and } b) \)
    - Example: 
      ![Diagram](image)
      - \( \text{ULPErr}(a, b) \leq 2 \cdot \text{RelErr}(a, b)/\epsilon \)
## Results

<table>
<thead>
<tr>
<th></th>
<th>Interval</th>
<th>Bound on ULP error</th>
<th># of intervals</th>
<th># of δ's</th>
<th>Size of “gaps”</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>[−4, 4]</td>
<td>14</td>
<td>13</td>
<td>29</td>
<td>36</td>
</tr>
<tr>
<td>sin</td>
<td>$\left[ -\frac{\pi}{2}, \frac{\pi}{2} \right]$</td>
<td>9</td>
<td>33</td>
<td>53</td>
<td>110</td>
</tr>
<tr>
<td>log</td>
<td>$(0, 4) \setminus \left[ \frac{4095}{4096}, 1 \right)$</td>
<td>$1 \times 10^{14}$</td>
<td>$2^{21}$</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$\left[ \frac{4095}{4096}, 1 \right)$</td>
<td></td>
<td>1</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Function</td>
<td>Interval</td>
<td>Bound on ULP error</td>
<td># of intervals</td>
<td># of $\delta$'s</td>
<td>Size of “gaps”</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>$\exp$</td>
<td>$[-4,4]$</td>
<td>14</td>
<td>13</td>
<td>29</td>
<td>36</td>
</tr>
<tr>
<td>$\sin$</td>
<td>$[-\frac{\pi}{2}, \frac{\pi}{2}]$</td>
<td>9</td>
<td>33</td>
<td>53</td>
<td>110</td>
</tr>
<tr>
<td>$\log$</td>
<td>$(0,4) \setminus \left[\frac{4095}{4096}, 1\right]$</td>
<td>21</td>
<td>$2^{21}$</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$\left[\frac{4095}{4096}, 1\right)$</td>
<td>$1 \times 10^{14}$</td>
<td>1</td>
<td>25</td>
<td>0</td>
</tr>
</tbody>
</table>

best illustrates the power of our method
Results: $\sin, \log$

- **sin**
  - **y-axis:** ULP error
  - **x-axis:** input value
  - **Orange line:** bounds on the intervals
  - **Blue dots:** errors on the “gaps”

- **log**
  - **y-axis:** value
  - **x-axis:** input value
  - **Orange line:** bounds on the intervals
  - **10^{14}:** value
**Results:** $\sin, \log$
Limitations of Our Method

• May construct a large number of intervals
  • Example: \(0x5fe6eb50c7b537a9 \rightarrow (x >> 1)\)
  • For this example, our method constructs \(2^{63}\) intervals
Limitations of Our Method

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

- May produce loose error bounds
  - Example: $10^{14}$ ULPs for $\log$ on $[\frac{4095}{4096}, 1)$
  - 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?