Icing
Supporting Fast-Math Style Optimizations in a Verified Compiler

Heiko Becker, Eva Darulova, Magnus Myreen, Zachary Tatlock
How we develop programs

readability over performance
How we develop programs

readability over performance

compiler should make program fast
How we develop programs

readability over performance

Compiler optimizations are a vital part of the development process

make program fast
What does gcc's ffast-math actually do?

I understand gcc's `--ffast-math` flag can greatly increase speed for float ops, and goes outside of IEEE standards, but I can't seem to find information on what is really happening when it's on. Can anyone please explain some of the details and maybe give a clear example of how something would change if the flag was on or off?

I did try digging through S.O. for similar questions but couldn't find anything explaining the workings of ffast-math.
The need for understandable optimizations

What does gcc's **ffast-math** actually do?

I understand gcc's **--ffast-math** flag can greatly increase speed for float ops, and goes outside of IEEE standards, but I can't seem to find information on what is really happening when it's on. Can anyone please explain some of the details and maybe give a clear example of how something would change if the flag was on or off?

I did try digging through S.O. for similar questions but couldn't find anything explaining the workings of ffast-math.

**Tags**: performance, math, gcc, floating-point, fast-math

**Asking details**: asked 7 years, 10 months ago, viewed 41,233 times, active 10 months ago
The need for understandable optimizations

What does gcc's ffast-math actually do?

I understand gcc's `--ffast-math` flag can greatly increase speed for float ops, and goes outside of IEEE standards, but I can't seem to find information on what is really happening when it's on. Can anyone please explain some of the details and maybe give a clear example of how something would change if the flag was on or off?

I did try digging through S.O. for similar questions but couldn't find anything explaining the workings of ffast-math.
The long road of compiler verification

1967: McCarthy, Painter;
   Correctness of a Compiler for Arithmetic Expressions (Integer’s only)

2009: Leroy;
   Formal Verification of a Realistic Compiler

2019: Lööw et al;
   Verified Compilation on a Verified Processor
The long road of compiler verification

1967: McCarthy, Painter;
Correctness of a Compiler for Arithmetic Expressions (integer’s only)

2009: Leroy;
Formal Verification of a Realistic Compiler

2019: Lööw et al;
Verified Compilation on a Verified Processor
The long road of compiler verification

**1967**: McCarthy, Painter;
Correctness of a Compiler for Arithmetic Expressions (Integer’s only)

**2009**: Leroy;
Formal Verification of a Realistic Compiler

**2019**: Lööw et al;
Verified Compilation on a Verified Processor
The long road of compiler verification

1967: McCarthy, Painter;
Correctness of a Compiler for Arithmetic Expressions (Integer’s only)

2009: Leroy;
Formal Verification of a Realistic Compiler

2019: Lööw et al;
Verified Compilation on a Verified Processor
The long road of compiler verification

1967: McCarthy, Painter;
Correctness of a Compiler for Arithmetic Expressions (Integer’s only)

2009: Leroy;
Formal Verification of a Realistic Compiler

2015: Boldo et al.;
Verified Compilation of Floating-Point Programs

2019: Lööw et al;
Verified Compilation on a Verified Processor
The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ....)  Verified Compilers (CakeML, ...)

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ....)  Verified Compilers (CakeML, ...)

• apply aggressive optimizations
The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ....)  Verified Compilers (CakeML, ...)

- apply aggressive optimizations
- do not preserve IEEE754 semantics
The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ....) Verified Compilers (CakeML, ...)

• apply aggressive optimizations
• do not preserve IEEE754 semantics
• give no guarantees on the result
## The state-of-the-art for fast-math

<table>
<thead>
<tr>
<th>Unverified Compilers (gcc, clang, ....)</th>
<th>Verified Compilers (CakeML, ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• apply aggressive optimizations</td>
<td>• apply no floating-point optimizations</td>
</tr>
<tr>
<td>• do not preserve IEEE754 semantics</td>
<td></td>
</tr>
<tr>
<td>• give no guarantees on the result</td>
<td></td>
</tr>
</tbody>
</table>
### The state-of-the-art for fast-math

<table>
<thead>
<tr>
<th>Unverified Compilers (gcc, clang, ....)</th>
<th>Verified Compilers (CakeML, ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• apply aggressive optimizations</td>
<td>• apply no floating-point optimizations</td>
</tr>
<tr>
<td>• do not preserve IEEE754 semantics</td>
<td>• fully preserve IEEE754 semantics</td>
</tr>
<tr>
<td>• give no guarantees on the result</td>
<td></td>
</tr>
</tbody>
</table>
The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ....)

• apply aggressive optimizations
• do not preserve IEEE754 semantics
• give no guarantees on the result

Verified Compilers (CakeML, ...)

• apply no floating-point optimizations
• fully preserve IEEE754 semantics
• guarantee preserving literal meaning
The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ....)  Verified Compilers (CakeML, ...)

- apply aggressive optimizations
- do not preserve IEEE754 semantics
- give no guarantees on the result

We need a semantics to handle fast-math optimizations in verified compilers
Contributions

Icing, a nondeterministic semantics for floating-points:

- Support for subset of gcc’s fast-math optimizations
- Optimization with fine-grained control
- Implementation of three optimizers
- Verification in HOL4
- Connection to CakeML
Optimizations in Icing

Example Optimizations:

\[
\begin{align*}
  a + b & \quad \rightarrow \quad b + a \\
  a \times b & \quad \rightarrow \quad b \times a \\
  a + (b + c) & \quad \rightarrow \quad (a + b) + c \\
  a \times (b \times c) & \quad \rightarrow \quad (a \times b) \times c \\
  a \times b + c & \quad \rightarrow \quad FMA(a, b, c)
\end{align*}
\]
Optimizations in Icing

Example Optimizations:

Commutativity (preserves IEEE754)

- $a + b \rightarrow b + a$
- $a \times b \rightarrow b \times a$
- $a + (b + c) \rightarrow (a + b) + c$
- $a \times (b \times c) \rightarrow (a \times b) \times c$
- $a \times b + c \rightarrow FMA(a, b, c)$
Optimizations in Icing

Example Optimizations:

- $a + b \rightarrow b + a$
- $a \times b \rightarrow b \times a$
- $a + (b + c) \rightarrow (a + b) + c$
- $a \times (b \times c) \rightarrow (a \times b) \times c$
- $a \times b \times c \rightarrow FMA(a,b,c)$

Source $s$ → Target $t$

Associativity (no IEEE754)
Optimizations in Icing

Example Optimizations:

- $a + b \rightarrow b + a$
- $a \times b \rightarrow b \times a$
- $a + (b + c) \rightarrow (a + b) + c$
- $a \times (b \times c) \rightarrow (a \times b) \times c$
- $a \times b + c \rightarrow FMA(a, b, c)$
Optimizations in Icing

Example Optimizations:

- $a + b \rightarrow b + a$
- $a \times b \rightarrow b \times a$
- $a + (b + c) \rightarrow (a + b) + c$
- $a \times (b \times c) \rightarrow (a \times b) \times c$
- $a \times b + c \rightarrow FMA(a, b, c)$
Floating-Point Values in Icing

IEEE754:  
3.5 + 2.0 = 5.5

Icing:
Floating-Point Values in Icing

IEEE754:   Icing:

3.5 + 2.0 = 5.5 ← floating-point word
Floating-Point Values in Icing

IEEE754: 3.5 + 2.0 = 5.5 ← floating-point word

Icing: 3.5 + 2.0 =
Floating-Point Values in Icing

IEEE754:

3.5 + 2.0 = 5.5

Icing:

3.5 + 2.0 =

\[
\begin{array}{c}
3.5 \\
+ \\
2.0
\end{array}
\]
Floating-Point Values in Icing

IEEE754:

\[ 3.5 + 2.0 = 5.5 \text{ } \text{floating-point word} \]

Icing:

\[ 3.5 + 2.0 = \]

value tree for addition
Floating-Point Values in Icing

IEEE754:

3.5 + 2.0 = 5.5 ← floating-point word

3.5 + (2.0 + 1.5) = 12.25

Icing:

3.5 + 2.0 =

value tree for addition
Floating-Point Values in Icing

IEEE754:

\[ 3.5 + 2.0 = 5.5 \]  

floating-point word

\[ 3.5 + (2.0 + 1.5) = 12.25 \]

Icing:

\[ 3.5 + 2.0 = \]

\[ + \]

\[ 3.5 \]

\[ 2.0 \]

value tree for addition

\[ 3.5 + (2.0 + 1.5) = \]
Floating-Point Values in Icing

IEEE754:

3.5 + 2.0 = 5.5 ← floating-point word

3.5 + (2.0 + 1.5) = 12.25

Icing:

3.5 + 2.0 =

3.5 + (2.0 + 1.5) =

value tree for addition
Floating-Point Values in Icing

IEEE754:

3.5 + 2.0 = 5.5

3.5 + (2.0 + 1.5) = 12.

Icing:

3.5 + 2.0 = 8

3.5 + (2.0 + 1.5) = 12.
Icing’s semantics

<table>
<thead>
<tr>
<th>Allowed Optimization:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \times b + c$</td>
<td>$FMA(a, b, c)$</td>
</tr>
<tr>
<td>$a \times b$</td>
<td>$b \times a$</td>
</tr>
</tbody>
</table>

$\text{opt:}(x \times 2.4 + y)$
Allowed Optimization:

\[ a \times b + c \rightarrow FMA(a, b, c) \]
\[ a \times b \rightarrow b \times a \]

Included in the semantics

\[ \text{opt}:(x \times 2.4 + y) \]
Icing’s semantics

Allowed Optimization:

\[ a \times b + c \rightarrow FMA(a, b, c) \]
\[ a \times b \rightarrow b \times a \]

Includes in the semantics:

fine-grained control \rightarrow \text{opt:}(x \times 2.4 + y)
Allowed Optimization:

\[ a \times b + c \longrightarrow FMA(a, b, c) \]

\[ a \times b \longrightarrow b \times a \]

Included in the semantics

\[ (x \times 2.4 + y) \]

fine-grained control \[\rightarrow\ opt:(x \times 2.4 + y)\]
Icing’s semantics

Allowed Optimization:

\[ a \times b + c \rightarrow FMA(a, b, c) \]
\[ a \times b \rightarrow b \times a \]

Included in the semantics

\[ \text{fine-grained control} \rightarrow \text{opt:}(x \times 2.4 + y) \]

\[
\begin{align*}
\text{opt:}(x \times 2.4 + y) & \rightarrow FMA(x, 2.4, y) \\
& \rightarrow x \times 2.4 + y \\
& \rightarrow x \times 2.4 + y \\
& \rightarrow x \times 2.4 + y \\
& \rightarrow x \times 2.4 + y
\end{align*}
\]
Icing’s semantics

Allowed Optimization:

\[ a \times b + c \rightarrow FMA(a, b, c) \]

Included in the semantics:

\[ a \times b \rightarrow b \times a \]

Icing: a direct fit for fast-math with fine-grained control and support for different optimizations
Modelling the state-of-the-art

Unverified Compilers (gcc, clang, ....)

- aggressive optimizations
- no IEEE754 semantics
- no guarantees on the result

Verified Compilers (CakeML, ...)

- no floating-point optimizations
- IEEE754 semantics
- introduces no new behaviour

Icing provides:

- greedy optimizer
- IEEE754 Translator
Icing provides:

**Unverified Compilers (gcc, clang, ....)**

- aggressive optimizations
- no IEEE754 semantics
- no guarantees on the result

**Verified Compilers (CakeML, ...)**

- no floating-point optimizations
- IEEE754 semantics
- introduces no new behaviour

---

**Modelling the state-of-the-art**

- **Icing provides:**
  - greedy optimizer
  - IEEE754 Translator
What can we prove about the optimizers

Greedy optimizer:

⊢ if evaluating the greedily optimized program \( p \) returns \( \nu \)
then \( \nu \) is a possible result of evaluating \( p \) with the optimizations of the greedy optimizer

IEEE754 translator:

⊢ after running the IEEE754 translator on program \( p \) no optimizations can be applied by Icing semantics

⊢ after running the IEEE754 translator on program \( p \) Icing semantics are deterministic no matter which optimizations are allowed

The greedy optimizer applies optimizations with respect to Icing semantics

The IEEE754 translator preserves literal meaning (like CompCert/CakeML)
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

\[ \times \ast (y + z) \]
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

\[ x \times (y + z) \]

\[ x \times y + x \times z \]

Compiler
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

\[ x \times (y + z) \]

Compiler

Semantics

\[ x1 \times y + x2 \times z \]
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

\[ x \times (y + z) \]

Compiler

\[ x \times y + x \times z \]

Semantics

\[ x_1 \times y + x_2 \times z \]

What if these do not match?
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

What if these do not match?
Distributivity in Icing

$a \times (b + c)$ \rightarrow $a \times b + a \times c$

Semantics

$x' \times (y + z)$

Compiler

$x \times y + x \times z$

$x'$ rewrites into $x_1$ and $x_2$

Semantics

$x_1 \times y + x_2 \times z$

What if these do not match?
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

What if these do not match?
Distributivity in Icing

\[ a \times (b + c) \rightarrow a \times b + a \times c \]

Semantics: 
\[ x' \times (y + z) \]

Compiler: 
\[ x \times y + x \times z \]

Semantics: 
\[ x1 \times y + x2 \times z \]

What if these do not match?

Conditionals: Tricky! (see paper)
Enable fast-math mode. This defines the `FAST_MATH` preprocessor macro which makes assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. \( *c == a * c + b * c \)),
- Operands to floating-point operations are not equal to \( NaN \) and \( Inf \), and
- \( +0 \) and \( -0 \) are interchangeable.
Enable fast-math mode. This defines the `FAST_MATH` preprocessor macro.

Lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `a * c == a * c + b * c`),
- Operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.
Handling more of gcc’s rewrites

Enable fast-math mode. This defines the `FAST_MATH` preprocessor macro.

- Lossy assumptions about floating-point math. These include:
  - Floating-point math obeys regular algebraic rules for real numbers (e.g. `c == a * c + b * c`),
  - Operands to floating-point operations are not equal to `NaN` and `Inf`, and
  - `+0` and `-0` are interchangeable.

Official clang documentation
Enable fast-math mode. This defines the `FAST_MATH` preprocessor macro, making assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`).
- Operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.
Handling more of gcc’s rewrites

Precondition allows to check condition before applying a rewrite

Enable fast-math mode. This defines the _FAST_MATH_ preprocessor macro.

lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. 
  \( a * c == a * b * c \)),
- operands to floating-point operations are not equal to NaN and Inf, and
- +0 and -0 are interchangeable.

Official clang documentation:

gcc: isNaN (c) → F
Handling more of gcc’s rewrites

Precondition allows to check condition before applying a rewrite

Enable fast-math mode. This defines the `FAST_MATH` preprocessor macro lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `c == a * c + b * c`),
- Operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation
Handling more of gcc’s rewrites

Precondition allows to check condition before applying a rewrite

Enable fast-math mode. This defines the `FAST_MATH` preprocessor macro.

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `c == a * c + b * c`),
- Operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `−0` are interchangeable.

Official clang documentation

Only $NaN$ is unequal to itself.
How can the preconditions be checked

- Roundoff Errors
  - FPTaylor [TOPLAS March 19]
- Exceptions
  - Gappa [SAC ‘06]
- Global Range Bounds
  - SMT-solvers (Z3 [TACAS ‘08], ...)
- Daisy [TACAS ‘18]
- Verasco [POPL ‘15]
Icings interface to external tools

Discharge checks in-place

\[ a, b, c \text{ variables} \Rightarrow a \times (b + c) \Rightarrow a \times b + a \times c \]

simple local check

\[ \Rightarrow \text{ checked before applying optimization} \]

Record assumed proposition

\[ c = c \Rightarrow \text{isNaN}(c) \Rightarrow False \]

complex global property

\[ \Rightarrow \text{ checked offline after compiling} \]
What does gcc’s fast-math actually do?

Nondeterministic Icing (with optimizations)

Deterministic Icing (without optimizations)

CakeML source

Outlook:
• integrate with external tools
• verify larger optimizations
• integrate into CakeML semantics