Defining -ffast-math is hard!

Doc. No: P3875R0
Contact: Hans Boehm (hboehm@google.com)
Audience: SG6
Date: Oct 6, 2025

Abstract

We argue that defining -ffast-math transformations sufficiently precisely to be useful in a standard is hard, and perhaps too hard to be feasible.

Introduction

This is a slight expansion of previous discussions, either on the SG6 reflector, or as part of prior presentations, There has been discussion in SG6 of explicitly allowing non-value-preserving floating-point optimizations, along the lines of optimizations normally enabled by -ffast-math, though possibly enabled more selectively via type-wrappers..

Many of the commonly supported relaxations appear to me to be very difficult to describe precisely, By "precisely", I mean sufficiently that a user could preclude strange interactions between compiler optimizations, or reason precisely about program correctness.

In a sense, telling the compiler that it may, for example, reassociate floating-point expressions amounts to allowing the optimizer to assume the false hypothesis that two unequal expressions are equal. Though this in a sense allows the compiler to generate arbitrary code, this commonly works out OK, and often proves useful in practice, in that compiler writers apply this license judiciously, thus usually preventing egregiously incorrect results. But it is very unclear to me whether these “judicious application” rules can be specified sufficiently robustly or unambiguously for a language standard..

This is closely related to Joshua Cranmer’s earlier papers. P3714 makes a similar argument, focussed on one particular attempt at semantics. P3715 provides much helpful background.

Interaction with other optimizations

We focus on reassociation, though any transformation that doesn't preserve exact floating-point values can be expected to have a similar effect. What does it mean to allow reassociation? Given

```
inline float f(float x, float y, float z) {
  return x + y + z;
}
```

can the compiler inline `f` in two places and associate the inlined version differently, so that a "pure function" returns two different results for the same arguments?

If we have

```
w = f(a, b, c);
use w;
...
use w again;
```

And the compiler needs to spill w during the ellipsis, can it instead recompute `w`, with a different association, so that `w` appears to have two slightly different values at different times? The reflector discussion with Joshua Cranmer points out that this seems to be commonly allowed.

If we compute the same value at different times with different associations, can the compiler assume the results must be the same and generate a completely bogus answer if not? Clearly the intent here is no, but how do you specify that?

As a specific example of this, consider the following code snippet:

```
if (a + b + c > 0.0) {
  x = 1.0;
  longer code sequence;
  y *= 2.0;
} else {
  same longer code sequence;
}
```

If we are trying to minimize code size, we must presumably disallow optimizing this to

```
if (a + b + c > 0.0) {
  x = 1.0;
}
longer code sequence;
if (a + (b + c) > 0.0) {
  y *= 2.0;
```

```
}
```

since this transformed version can execute exactly one of the two assignments to x and y respectively. That's clearly not consistent with the intent of the original code. Yet the transformation may make sense if registers are in short supply and a + (b + c) is available as a result of executing the "longer code sequence". And it results from a sequence of code transformations which is entirely benign, except for the reassociation.

Even if we were willing to specify the optimization rules as a set of allowable program transformations, I think it would be difficult to preclude such an obviously undesirable transformation sequence. And we do not want to do that, since it precludes new optimizations or nontraditional implementation styles.

Clearly we can easily construct a similar example for FMA contraction, rather than reassociation.

Flush-to-zero library interactions

Flush-to-zero can produce similarly problematic results, because it is typically not applied to all generated floating-point values. This is compounded by the facts that, as pointed out in P3715, it is often set in a different compilation unit and thus not compiler visible. Furthermore, a few library functions no longer have clean semantics in the presence of flush-to-zero. Here's an example I found particularly counterintuitive:

```
int main() {
 double smallest = nextafter(0.0, 1.0);
 if (smallest > 0.0) {
   printf("Positive: %g\n", smallest);
 } else {
   printf("Wrong: %g\n", smallest);
 }
 return 0;
}
```

With clang 16 -ffast-math it produces:

```
Wrong: 4.94066e-324
```

Note that the comparison result is inconsistent with the printed value. Apparently `nextafter()` continues to produce the smallest positive denormalized double value, and that gets printed, but comparison flushes to zero. This kind of behavior seems rather difficult to define correctly, or to explain to a user unfamiliar with the full implementation details. It is no longer clear that nextafter() can be cleanly defined: Denormal values can still be generated in some contexts, but

not others. There is no longer a well-defined set of "representable values”.