Virtual values have virtual value

Document number: P3714R0
Date: 2025-05-19
Author: Joshua Cranmer joshua.cranmer@intel.com
Audience: SG6

P3565 proposes a model of floating-point semantics that is based on the idea of virtual values, which essentially boils down to three propositions. First, the result of a floating-point operation is a virtual value that could be more precise than any value representable in the declared type of the value. Second, that some operations (which ones are in this class are not yet fully decided) would definitively fix the value to one representable in the declared type. Third, that all uses of any value would necessarily see the same value. The goal of this proposal is to model x87 excess precision and floating-point contraction. Since I do not think x87 floating-point is a good use of committee time, I will focus on floating-point contraction.

The properties above boil down to arguing that the assertion in the following code must never fail under any possible set of circumstances (including code being in a different translation unit):

```
float sum(float a, float b) { return a + b; }
void test(float x, float y, float z) {
  float mul = x * y;
  assert(mul + z == sum(mul, z));
}
```

While this is an appealing property for users, it is a very difficult, maybe impossible, property for the optimizer to guarantee (except in the trivial case of "never contract"). This is for several reasons.

First, this ties the legality of an optimization at one phase in the compiler to the decisions it will make in a later phase. For example, the compiler may constant-fold the first expression, but whether or not it would need to use an FMA or an unfused expression is dependent on its future ability to fuse the second expression. Whatever decision it makes (short of doing nothing), there is a version of the second expression which would retroactively make that decision wrong.

Second, source-level data dependencies cannot be reliably preserved by a compiler. Redundancy elimination optimizations routinely remove dependencies that previously existed and create new dependencies that didn't exist (when b == c, uses of b may be replaced with c or vice versa). The inability to preserve these dependencies has previously torpedoed both `memory_order_consume` and pointer provenance.

Third, the effect of this behavior is to introduce implicit optimization barriers based on the presence of side-effect-free code, and the history of compilers is that such implicit optimization barriers will be frequently dropped in practice.

Ultimately, P3565 appears to provide virtually no value for implementers. The C standard already has mechanisms to support the concerns in the paper (`FLT_EVAL_METHOD` and `#pragma STDC FP_CONTRACT`), and these mechanisms have well-prescribed points that impose barriers that prevent the optimization here, just as P3565 wants to prescribe (albeit in different locations).

However, implementers today do not observe these barriers correctly, especially when driven by command-line flags. Thus their behavior is nonconforming today, and would remain nonconforming even if this paper is adopted.