# “Undefined behavior” and the concurrency memory model

Doc. No: P2215R1
Contact:Hans Boehm  hboehm@google.com)
Audience: SG1 (mainly), SG12
Date: Sept. 14, 2020
Target: Eventually C++2X, ideally X=3

## Abstract

Previous SG1 discussions of out-of-thin-air behavior seemed to regularly bump into questions of what exactly “undefined behavior” means in a concurrent context. Our current out-of-thin-air issues strongly interact with undefined behavior, and are strongly aggravated by it, as P1916 points out. This is an initial attempt at, informally for now, specifying what “undefined behavior” should really mean in this context.

## Introduction

This is a preliminary presentation refining a view of “undefined behavior” that I previously expressed to SG1. If we were to agree that this is abstractly the correct interpretation, then I believe we should add wording to that effect. But it is still too early for precise wording.

The current working paper states (in a note in 4.1.1 [intro.abstract] p4): “This document imposes no requirements on the behavior of programs that contain undefined behavior.”  There are similar statements elsewhere in the standard. This means that if a program contains undefined behavior, it is allowed to behave in unexpected ways even before the undefined behavior is encountered. Optimizers rely on this freedom, in that it allows them to assume that programs will not produce undefined behavior, and use that to e.g. infer that certain program paths are impossible.

But there are limits to such “retroactive” undefined behavior. We cannot allow causal cycles in which undefined behavior (UB) is introduced by the same UB. If we have:

```
int a;
int *p;

int main() {
```

```
  p = &a;
  *p = 17;
  return 0;
}
```

We clearly do not allow *p = 17 to produce UB, whose effect is to assign a bad value to p, thus causing the UB. The initial undefined behavior cannot be caused by iself.

For single-threaded programs, this is probably too obvious to belabor. For multithreaded programs, this becomes far more subtle. It is clear that if a valid execution includes UB, then the program should be able to generate any observable behavior. But what constitutes a valid execution with UB? Such an execution should not involve UB caused by itself. However it's well-known that "caused by" is hard to pin down in multi-threaded code, mostly because compilers routinely eliminate what syntactically look like dependencies.

A valid execution requires the execution of each of the threads to be consistent, and it requires that loads and stores from/to shared locations to be consistent. But what does the latter condition mean when one of the thread executions under consideration includes UB?

## A simple interpretation of UB

One possible interpretation is to say that UB does not play in the memory model at all. As far as the memory model is concerned, UB behavior terminates execution without affecting memory contents at all. Then, effectively after the fact, we assign arbitrary behavior to any program that is so terminated. Let's call this

**Interpretation A**:

1. For the purposes of the concurrency memory model, UB has no effect at all. It's a no-op. (It doesn't really matter what happens afterwards, so we can also view it as equivalent to _exit() in the memory model.)

2. If there is a valid execution on a given input that includes UB anywhere, or includes a data race, then the program on that input can have any observable behavior whatsoever.

## Accommodating existing implementations

Interpretation A seems like a reasonable model to me. However, P1916 points out that it is not consistent with current implementations. Consider the following example, adapted slightly from the "tweaked version" of the first example there:

Consider the following example where x and y have type atomic<int>, and are initially zero:

| Thread 1 | Thread 2 |
|---|---|
| `r1 = x.load(memory_order_relaxed);`<br>`if (r1 == 0) {`<br>`  y.store(1, memory_order_relaxed);`<br>`} else {`<br>`  <undefined behavior>`<br>`}` | `// x = y`<br>`r2 = y.load(memory_order_relaxed);`<br>`x.store(r2, memory_order_relaxed);` |

As pointed out in P1916, without the else clause (the blue text), r1 == 1 would be impossible. It would imply r2 == 1, which would make the execution of the else-clause impossible. Hence, if the core concurrency memory model assigned no semantics to UB, as in Interpretation A, the else clause could never be executed, and hence the entire program would have well-defined semantics, with r1 == 1 still impossible.

P1916 shows that in reality, compilers will deduce that the else clause is not executed (since if it were, any behavior is allowed, and hence we are allowed to mis-compile the program), and hence treat the store to y as unconditional. This cannot be explained with Interpretation A, which assigns no additional semantics to UB in an unexecuted else clause. We thus propose

**Interpretation B:**

1. For the purposes of the concurrency memory model, UB is treated as potentially atomically storing any values whatsoever into any location, with any memory order. (But see next section for possible constraints on the memory order.)
2. If there is a valid execution on a given input that includes UB anywhere, or includes a data race, then the program on that input can have any observable behavior whatsoever.

This correctly mirrors the fact that “undefined behavior” often reflects implementation behavior that takes a wild branch or stores to an arbitrary “out of bounds” location.

Note that implicit UB store operations perform atomic stores; ordinary loads always load from stores that happen before them. Thus the only way that UB-implied stores can affect the execution that led to the UB is via atomic loads. The implied store could also add data races with ordinary loads, but this wouldn’t affect whether the UB is possible, and hence also would have no effect on the final observable behavior.

With this view of UB, the previous example is allowed to behave as

| Thread 1 | Thread 2 |
| --- | --- |
| `r1 = x.load(memory_order_relaxed);`<br>`if (r1 == 0) {`<br>`  y.store(1, memory_order_relaxed);`<br>`} else {`<br>`  y.store(1, memory_order_relaxed);`<br>`}` | `// x = y`<br>`r2 = y.load(memory_order_relaxed);`<br>`x.store(r2, memory_order_relaxed);` |

This makes the store to y unconditional, allowing it to be reordered with the load from x.

## Interaction with acquire/release and `memory_order_load_store`

P1217 suggests adding `memory_order_load_store` which. Informally, a `memory_order_load_store` load may not be reordered with a subsequent `memory_order_load_store` store. It serves as a memory_order_relaxed replacement that is slightly more expensive, but provably avoids out-of-thin-air or read-from-unexecuted-branch results.

We would like this to also provably avoid out-of-thin-air results in the presence of undefined behavior. Thus if again x and y are initially zero, then

| Thread 1 | Thread 2 |
| --- | --- |
| `r1 = x.load(memory_order_load_store);`<br>`if (r1 == 1) {`<br>`  <undefined behavior>`<br>`}` | `r2 = y.load(memory_order_load_store);`<br>`if (r2 == 1) {`<br>`  <undefined behavior>`<br>`}` |

should guarantee `r1 = r2` = 0 and not result in undefined behavior. As stated, it currently still allows UB, since the undefined-behavior-implied stores may be `memory_order_relaxed`.

In fact, we have the same issue with `memory_order_acquire`, or even `memory_order_seq_cst` as with `memory_order_load_store.`

The root of the problem is that we are allowing “undefined behavior” stores to be advanced earlier in the thread. Intuitively, these stores should not be able to advance to before the behavior that caused them. We thus propose:

**Interpretation B':**

1. For the purposes of the concurrency memory model, UB is treated as potentially atomically storing any values whatsoever into any location, *with* `memory_order_release`.
2. If there is a valid execution on a given input that includes UB anywhere, or includes a data race, then the program on that input can have any observable behavior whatsoever.

and hereby invite others to poke holes into this formulation.

# Consequences

David Goldblatt correctly points out that we have assumed that UB behavior is associated with some specific program evaluation, so that we know where implicit stores might be introduced. We believe that is unavoidable, and we need to ensure that the standard consistently specifies this.

All of our proposed specifications avoid UB initiated through causal cycles in fully sequenced single-threaded programs. More strongly, if A happens before B, then there is no danger that a load in A sees an explicit or implicit store in B, so there is no way, without the aid of another thread or unsequenced code, that B can impact the value for a load in A.

If the compiler can prove that every execution of A is always followed by an execution of B that will include undefined behavior, then it is still entitled to miscompile A. But this will not affect the overall semantics of the program, since every execution of A will result in UB. In this sense, UB can still affect earlier parts of the program. But, as expected, conditional undefined behavior cannot be reordered with a load that might impact the condition.The compiler can leverage potential undefined behavior to exclude certain paths in its analyses, but it cannot speculate such undefined behavior.

We believe that interpretation B' leads to expected results for something like (again with x and y initially zero):

| Thread 1 | Thread 2 |
|---|---|
| `r1 = x.load(mo1);`<br>`a[r1] = 1;` | `r2 = y.load(mo2);`<br>`b[r2] = 1;` |

A mis-speculated load of x can, in most implementations, result in a store to y. Thus this should behave like:

| Thread 1 | Thread 2 |
| --- | --- |
| `r1 = x.load(mo1);`<br>`if (r1 != 0) {`<br>`  y.store(rand(),`<br>`  memory_order_release);`<br>`  <undefined behavior>`<br>`} else {`<br>`  a[0] = 1;`<br>`}` | `r2 = y.load(mo2);`<br>`if (r2 != 0) {`<br>`  x.store(rand(),`<br>`  memory_order_release);`<br>`  <undefined behavior>`<br>`} else {`<br>`  b[0] = 1;`<br>`}` |

In C++20 without the vague out-of-thin-air prohibition (and without `memory_order_consume`), either would allow out-of-thin-air behavior if and only if either `mo1` or `mo2` were `memory_order_relaxed`. In any real implementation, or with something like P1780, this would be prevented by compiler enforcement of the dependency between the load and the store.

Thus interpretation B' seems to reduce the OOTA problem with undefined behavior to the OOTA problem without undefined behavior, which was our goal.

## Revision history

R1 clarified that the memory_order_load_store ordering issue also applies to sequentially consistent loads. And it added the "Consequences" section, after discussions with Paul McKenney and David Goldblatt.