Defang and deprecate memory_order::consume

Doc. No: P3475R1
Contact: Hans Boehm (hboehm@google.com)
Audience: LEWG, CWG, LWG
Date: Jan 11, 2025
Target: C++26
(Arguably a revision of P0371R0. But P0371R1 took a different direction and was accepted.)

Abstract

We again propose to deprecate `memory_order_consume`. We suggested a different variant of this about 8 years ago in P0371R0, which SG1 did not like. Circumstances have changed, and SG1 seemed much more amenable to this at the St. Louis meeting.

History

R0 was approved without dissent in Wrocław, by both SG1 and EWG.
R1 added more detailed wording.

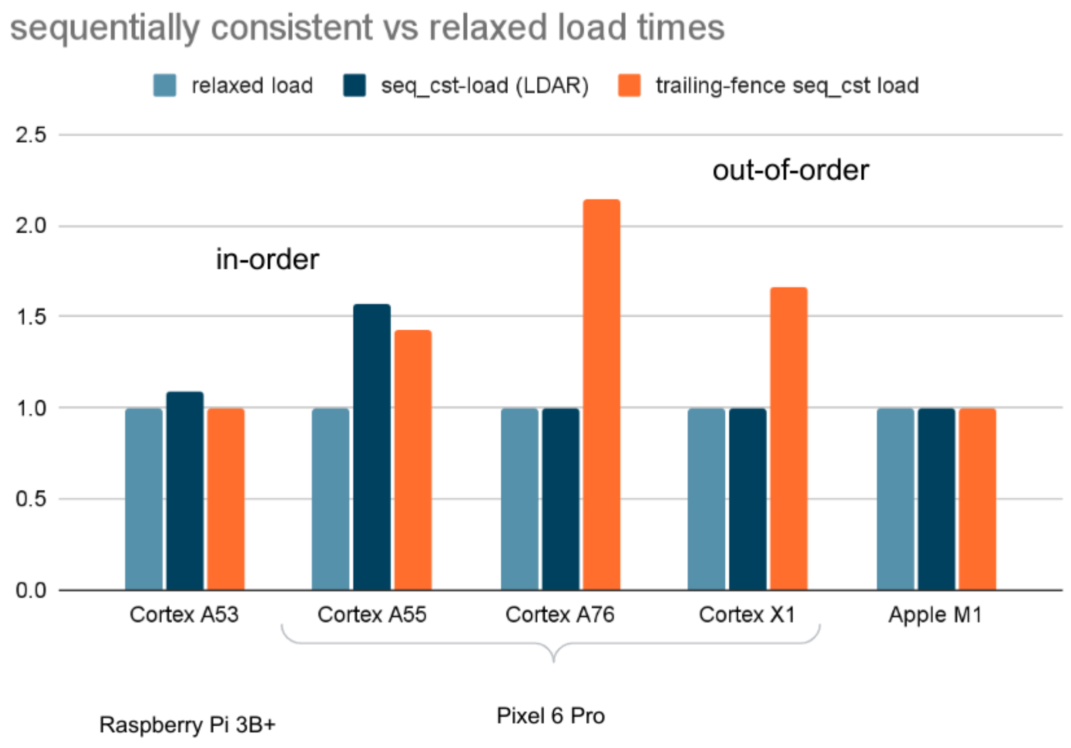
Rationale

It is widely accepted that the current definition of `memory_order::consume` in the standard is not useful. All current compilers essentially map it to `memory_order::acquire`. The difficulties appear to stem both from the high implementation complexity, from the fact that the current definition uses a fairly general definition of "dependency", thus requiring frequent and inconvenient use of the `kill_dependency()` call, and from the frequent need for `[[carries_dependency]]` annotations.

It is also widely accepted that `memory_order::consume` has frequent and important use cases in many large code bases. For example, the Linux kernel makes extensive use of RCU. To avoid over-constraining memory order for something like RCU on architectures like ARM and Power. something like `memory_order::consume` is required. Some core Android code similarly relies on dependency-based ordering.

On the other hand, there are strong arguments that we are not likely to introduce a new facility that can benefit significantly from the existing wording, and we should remove it:

1. This problem has been recognized for around a decade. We have had much discussion around it and possible replacements (e.g. P0098, P0190, P0750) We have not agreed on a replacement. We have not fixed even serious issues with its current specification, discouraging its use instead.
2. It complicates the memory model appreciably. More mathematical academic work tends to ignore its existence, since it is known to be broken. This makes it harder to translate such work into the standard. The memory model is in some need of tricky repair. It would be nice to avoid this unused complication.
3. The most widely-used CPU architectures no longer really require `memory_order::consume`. It was never very beneficial on X86. ARM provides hardware support for `memory_order::acquire` (and `memory_order::seq_cst`) loads that, for common micro-architectures and use-cases, perform similarly to `memory_order_relaxed`. RISC-V currently lacks this support, but appears to be moving in the same direction with the proposed Zalasr extension. The main architectures that could still benefit are Power and GPUs.
   Here is a comparison of microbenchmark `relaxed` vs `seq_cst` load times on some ARM implementations. (All versions include an indirect function call. so differences are understated, but not hugely so. The dark blue LDAR version uses the hardware `seq_cst` load instruction.)



4. In practice, non-portable, very careful, abuse of `memory_order_relaxed` seems to have carried the day for the remaining use cases.

Wording changes

Language section:

1. Change [intro.races] as follows:
    a) Remove [intro.races] p7 and p8, which define “carries a dependency” and “dependency-ordered before”
    b) Remove [intro.races] p9, the definition of “inter-thread happens before”.
    c) Remove [intro.races] p10, the definition of “happens before”.
    d) Modify [intro.races] to rename “simply happens before” to “happens before”

Some notes also require adjustment. The result is shown below, relative to N5001::

~~7 An evaluation A carries a dependency to an evaluation B if~~

> ~~(7.1) — the value of A is used as an operand of B, unless:~~
>
> > ~~(7.1.1) — B is an invocation of any specialization of std::kill_dependency (32.5.4), or~~
> >
> > ~~(7.1.2) — A is the left operand of a built-in logical and (&&, see 7.6.14) or logical or (||, see 7.6.15) operator, or~~
> >
> > ~~(7.1.3) — A is the left operand of a conditional (?:, see 7.6.16) operator, or~~
> >
> > ~~(7.1.4) — A is the left operand of the built-in comma (,) operator (7.6.20); or~~
> >
> > ~~(7.2) — A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or~~
>
> ~~(7.3) — for some evaluation X, A carries a dependency to X, and X carries a dependency to B.~~

~~[Note 7 : “Carries a dependency to” is a subset of “is sequenced before”, and is similarly strictly intra-thread. — end note]~~

~~8 An evaluation A is dependency-ordered before an evaluation B if~~

> ~~(8.1) — A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads the value written by A, or~~
>
> ~~(8.2) — for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.~~

~~[Note 8 : The relation “is dependency-ordered before” is analogous to “synchronizes with”, but uses release/consume in place of release/acquire. — end note]~~

~~9 An evaluation A inter-thread happens before an evaluation B if~~

> ~~(9.1) — A synchronizes with B, or~~
>
> ~~(9.2) — A is dependency-ordered before B, or~~
>
> ~~(9.3) — for some evaluation X~~
>
> > ~~(9.3.1) — A synchronizes with X and X is sequenced before B, or~~
> >
> > ~~(9.3.2) — A is sequenced before X and X inter-thread happens before B, or~~
> >
> > ~~(9.3.3) — A inter-thread happens before X and X inter-thread happens before B.~~

~~[Note 9 : The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced before”, “synchronizes with” and “dependency-ordered before” relationships, with two exceptions. The first exception is that a concatenation never ends with “dependency-ordered before” followed by “sequenced before”. The reason for this § 6.9.2.2 ©ISO/IEC 89 N5001 limitation is that a consume operation participating in a “dependency-ordered before” relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation never consist entirely of “sequenced before”. The reasons for this limitation are (1) to permit “inter-thread happens before” to be transitively closed and (2) the “happens before” relation, defined below, provides for relationships consisting entirely of “sequenced before”. — end note]~~

~~10 An evaluation A happens before an evaluation B (or, equivalently, B happens after A) if~~
~~(10.1) — A is sequenced before B, or~~
~~(10.2) — A inter-thread happens before B. The implementation shall ensure that no program execution demonstrates a cycle in the “happens before” relation.~~
~~[Note 10 : This cycle would otherwise be possible only through the use of consume operations. — end note]~~

11 An evaluation A ~~simply~~ happens before an evaluation B (or, equivalently, B happens after A) if either
(11.1) — A is sequenced before B, or
(11.2) — A synchronizes with B, or
(11.3) — A ~~simply~~ happens before X and X ~~simply~~ happens before B.
[Note 11 : ~~In the absence of consume operations, the happens before and simply happens before relations are identical~~. An evaluation cannot happen before itself. The final synchronizes-with relationship in such a cycle would not be possible, since it could only be introduced by an evaluation observing another evaluation that happens after it, something we never allow. — end note]

12 An evaluation A strongly happens before an evaluation D if, either
(12.1) — A is sequenced before D, or
(12.2) — A synchronizes with D, and both A and D are sequentially consistent atomic operations (32.5.4), or
(12.3) — there are evaluations B and C such that A is sequenced before B, B ~~simply~~ happens before C, and C is sequenced before D, or
(12.4) — there is an evaluation B such that A strongly happens before B, and B strongly happens before D.
[Note 12 : Informally, if A strongly happens before B, then A appears to be evaluated before B in all contexts. ~~Strongly happens before excludes consume operations.~~ — end note]

2. Remove [dcl.attr.depend], which defines `[[carries_dependency]]` in its entirety. There is not a reason to deprecate it, since remaining occurrences in code will presumably be ignored anyway:

~~9.12.4 Carries dependency attribute [dcl.attr.depend]~~

~~1 The attribute-token carries_dependency specifies dependency propagation into and out of functions. No attribute-argument-clause shall be present. The attribute may be applied to a parameter of a function or lambda, in which case it specifies that the initialization of the parameter carries a dependency to (6.9.2) each lvalue-to-rvalue conversion (7.3.2) of that object. The attribute may also be applied to a function or a lambda call operator, in which case it specifies that the return value, if any, carries a dependency to the evaluation of the function call expression.~~

~~2 The first declaration of a function shall specify the carries_dependency attribute for its declarator-id if any declaration of the function specifies the carries_dependency attribute. Furthermore, the first declaration of a function shall specify the carries_dependency attribute for a parameter if any declaration of that function specifies the carries_dependency attribute for that parameter. If a function or one of its parameters is declared with the carries_dependency attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the carries_dependency attribute in its first declaration in another translation unit, the program is ill-formed, no diagnostic required.~~

~~3 [Note 1 : The carries_dependency attribute does not change the meaning of the program, but might result in generation of more efficient code. — end note]~~

~~4 [Example 1 :~~

```
/* Translation unit A. */
struct foo { int* a; int* b; };
std::atomic foo_head[10];
int foo_array[10][10];
[[carries_dependency]] struct foo* f(int i) {
  return foo_head[i].load(memory_order::consume);
}

int g(int* x, int* y [[carries_dependency]]) {
  return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */
[[carries_dependency]] struct foo* f(int i);
int g(int* x, int* y [[carries_dependency]]);

int c = 3;
```

```
~~void h(int i) {~~
~~struct foo* p;~~

~~p = f(i);~~
~~  do_something_with(g(&c, p->a));~~
~~  do_something_with(g(p->a, &c));~~
~~}~~
```

~~The carries_dependency attribute on function f means that the return value carries a dependency out of f, so that the implementation need not constrain ordering upon return from f. Implementations of f and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences). Function g's second parameter has a carries_dependency attribute, but its first parameter does not. Therefore, function h's first call to g carries a dependency into g, but its second call does not. The implementation might need to insert a fence prior to the second call to g. — end example]~~

Library section

1. In [atomics.order], delete 1.3, which defines `consume`, but leave `consume` in the `enum class` as a placeholder, defining it in Annex D. I don't believe we can remove it, since we want to preserve the mapping to the underlying type. Specifically:

1 The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization order as defined in 6.9.2 and may provide for operation ordering. Its enumerated values and their meanings are as follows:
(1.1) — `memory_order::relaxed`: no operation orders memory.
(1.2) — `memory_order::release`, `memory_order::acq_rel`, and `memory_order::seq_cst`: a store operation performs a release operation on the affected memory location.
(1.3) — `memory_order::consume`: See appendix D.~~a load operation performs a consume operation on the affected memory location. [Note 1 : Prefer memory_order::acquire, which provides stronger guarantees than memory_order::consume. Implementations have found it infeasible to provide performance better than that of memory_order::acquire. Specification revisions are under consideration. — end note]~~
(1.4) — `memory_order::acquire`, `memory_order::acq_rel`, and `memory_order::seq_cst`: a load operation performs an acquire operation on the affected memory location.

[Note 2 : Atomic operations specifying `memory_order::relaxed` are relaxed with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. — end note]

2. Remove the following line from 32.5.2 [atomics.syn]:

```
    ~~inline constexpr memory_order memory_order_consume =~~
        ~~memory_order::consume; // freestanding;~~
```

and the following from 32.5.4 [atomics.order]:

~~(1.3) — memory_order::consume: a load operation performs a consume operation on the affected memory location.~~
~~[Note 1 : Prefer memory_order::acquire, which provides stronger guarantees than memory_order::consume. Implementations have found it infeasible to provide performance better than that of memory_order::acquire. Specification revisions are under consideration. — end note]~~

3. Remove mentions of `memory_oder::consume` from the many lists of acceptable memory orders in *32: Concurrency Support Library*. The changes all have the form:

`memory_order::relaxed,` ~~`memory_order::consume,`~~ `memory_order::acquire, or memory_order::seq_cst`

4. Remove the mention of memory_order::consume from [atomics.fences]:

```
extern "C" constexpr void atomic_thread_fence(memory_order order) noexcept;
```
5 Effects: Depending on the value of order, this operation:

(5.1) — has no effects, if `order == memory_order::relaxed`;
(5.2) — is an acquire fence, if `order == memory_order::acquire`~~`or order == memory_order::consume`~~;
(5.3) — is a release fence, if `order == memory_order::release`;
(5.4) — is both an acquire fence and a release fence, if `order == memory_order::acq_rel`;
(5.5) — is a sequentially consistent acquire and release fence, if `order == memory_order::seq_cst`.

4. Remove all mentions of `kill_dependency` from the atomics section, including the definition in the final paragraphs of [atomics.order].

In [atomics.syn]:

```
  inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
                                                         // freestanding
  inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
                                                         // freestanding
  ~~template<class T>~~
    ~~constexpr T kill_dependency(T y) noexcept; // freestanding~~
```

In [atomics.order]:

```
~~template<class T>~~
~~constexpr T kill_dependency(T y) noexcept;~~
```
~~12 *Effects:* The argument does not carry a dependency to the return value (6.9.2).~~
~~13 *Returns:* `y`.~~

5. Add a new section to Annex D:

D.? memory_order_consume [depr.consume]

`memory_order::consume` is allowed wherever `memory_order::acquire` is allowed, and it has the same memory-ordering effect.

```
template<class T> T kill_dependency(T y) noexcept; // freestanding
```
Returns: y.

<atomics> includes the following additional memory order declaration:

```
inline constexpr memory_order memory_order_consume =
    memory_order::consume; // freestanding
```