P0190R0: Proposal for New `memory_order_consume` Definition

Doc. No.: WG21/P0190R0
Date: 2016-01-13
Reply to: Paul E. McKenney, Michael Wong, Hans Boehm, and Jens Maurer
Email: paulmck@linux.vnet.ibm.com, michaelw@ca.ibm.com, boehm@acm.org, and Jens.Maurer@gmx.net

Other contributors:
Torvald Riegel, Jeff Preshing, Clark Nelson, Olivier Giroux, Lawrence Crowl, Alec Teal, David Howells, David Lang, George Spelvin, Jeff Law, Joseph S. Myers, Linus Torvalds, Mark Batty, Michael Matz, Peter Sewell, Peter Zijlstra, Ramana Radhakrishnan, Richard Biener, Will Deacon, Faisal Vali, Behan Webster, Tony Tye, JF Bastien, Thomas Koeppe, Boqun Feng, ...

January 19, 2016

This document is a follow-on to WG21/P0098R1,[1] based on email discussion and on discussions at the 2015 meeting at Kona, which should be consulted for background on `memory_order_consume` and for a number of alternatives to the definition in the standard [25]. The main purpose of `memory_order_consume` is to provide language support for read-copy update (RCU) [5, 10, 12, 13, 14, 16, 18], which is heavily used in the Linux kernel and which is seeing increasing use in user-level multi-threaded software [1, 2, 4, 7, 8, 21, 24]. In addition, the intersection of RCU and transactional memory is starting to garner significant attention [9, 11, 19, 20, 22].

However, this proposal has use cases beyond just RCU. For example, Java final-field accesses provide a closely related dependency guarantee, and a similar capability will likely be needed by any C++ garbage collector. Furthermore, garbage collectors enable use cases that are quite similar to those of RCU [6]. In addition, it is highly likely that additional concurrent code will rely on dependency ordering, given that mainstream implementations provide dependency-ordering guarantees via TSO or via hardware dependency ordering.

This document presents a proposed solution to the problems with `memory_order_consume` in the current C and C++ standards. Section 1 describes the overall approach, Section 2 provides an informal definition, Section 3 provides draft wording, Section 4 provides a series of litmus tests demonstrating dependency chains, and finally Section 5 summarizes benefits, drawbacks, and mitigations.

1 Approach

The purpose of `memory_order_consume` loads is to provide ordering guarantees similar to those of `memory_order_acquire` loads, but, in contrast to the wording in the current standard, only in cases where there is a *robust dependency* between the `memory_order_consume` load and some subsequent operation. A robust dependency is guaranteed to be preserved

[1] Or, in the near term, its predecessor, WG21/P0098R0 [17].

by both compilers and CPUs without the need for explicit memory-fence instructions,[2] and is a subset of the syntactic dependencies that earlier C++ standards [25] specify for `memory_order_consume`. In contrast, the approach currently in the standard requires the compiler to insert artificial dependencies or even memory-barrier instructions so that even non-robust dependencies will be preserved.[3]

Another conspicuous change is that this proposal does not require dependencies to be carried by integers, but instead only by pointers. Note that versions v4.2 and later of the Linux kernel avoid carrying dependencies through integer variables [15].[4]

In fact, the overall approach is to provide only those dependency-chain preservation guarantees that are actually used in recent releases of the Linux kernel. This has the beneficial effect of making a minimal implementation (excluding pointer-comparison intrinsics and diagnostics) trivial: The implementation need only compile a `memory_order_consume` load as if it were a `memory_order_relaxed` load.

2 Informal Dependency-Chain Definition

This section provides an informal description of the proposed solution. Section 2.1 lists requirements and desiderata, and Section 2.2 provides the informal description, examples, and discussion.

2.1 Requirements and Desiderata

There are a large number of constraints imposed on the problem of defining `memory_order_consume` dependency chains.

Many of them stem from the desire to accommodate existing dependency-chain practice, for example within the Linux kernel, but at the same time placing little or (preferably) no burden on implementations. Although invention does have its place, it should be noted that invention was exactly what resulted in an unworkable definition of `memory_order_consume` in the C++11 standard. Requirements stemming from Linux-kernel compatibility include:

1. Dependency chains must not depend solely on marking objects carrying dependencies. Objects include local variables, function parameters, and function return declarations.

2. Although it is perfectly acceptable for dependency chains to mark the load heading the chain, depedency chains must not depend on marking accesses or operations further down the chain.

3. Use of `memory_order_consume` loads should not result in unsolicited memory-fence instructions. High-quality implementations would avoid emitting such instructions, and not-so-high-quality implementations would at the very least be able to issue a warning when such an instruction was emitted.

4. Dependency chains need only be carried through pointer values, however, they must be carried to (not through) non-pointer values in a number of cases.

The Linux kernel requires that dependencies be carried through bit manipulations of pointer values, however, such bit manipulation invokes undefined behavior. Bit manipulation of pointer values is useful for tagging and for some types of memory allocators, and therefore might be worth standardizing in its own right.

Compiler writers require that dependency chains not require explicit tracing by implementations, such tracing being one of the major rocks on which the C++11 definition of `memory_order_consume` foundered.

Although Linux-kernel compatibility requires that dependency chains not depend solely on markings, it is entirely acceptable for such marking to provide additional benefits such as higher-quality diagnostics and software-engineering benefits. In fact,

[2] DEC Alpha is the memory-fence exception that proves the rule [3, 23].

[3] Note to Linux kernel hackers: From here on out, the word "implementation" will be used instead of "compiler", as is the custom in the C++ Standards Committee.

[4] That said, it may be necessary to provide extremely restricted dependencies through `intptr_t` and `uintptr_t` in order to support use of tag bits in pointers, which is required by a number of concurrent algorithms.

```
1 p = rcu_dereference(gp);
2 *p = 5;
```

Figure 1: Extending Dependency Chain on Left-Hand Side

the Linux kernel already provides the `__rcu` marking that may optionally be applied to RCU-protected pointers, that is, pointers that are loaded using `rcu_dereference()`, which is the Linux kernel's counterpart to a `memory_order_consume` load. One key hoped-for benefit of optional markings is ease of formal verification.

2.2 Informal Definition

This informal description covers operations that extend dependency chains (Section 2.2.1) and operations that terminate dependency chains (Section 2.2.2).

2.2.1 Extending Dependency Chains

The following categories of primitive operations extend dependency chains:[5]

1. Moving, copying, and casting.
2. Pointer offsets.
3. Dereferencing and address-of, including class-member access.
4. Miscellaneous operators.

Any other operation terminates a dependency chain. The operations that extend dependency chains are covered in more detail below.

Moving, Copying, and Casting: Values that are part of a dependency chain may be moved, copied, and casted (in some cases), and the dependency chain will propagate to the result.

[5] In case of operator overloading, the actual functions called must be analyzed in order to determine their effects on dependency chains.

1. If any pointer value is part of a dependency chain, then using that value as the left-hand side of an assignment expression extends the chain to cover the assignment. This rule is exercised in the Linux kernel by stores into fields making up an RCU protected data element. This is illustrated by Figure 1.
2. If any pointer value is part of a dependency chain, then using that value as the right-hand side of an assignment expression extends the chain to cover both the assignment and the value returned by that assignment statement. Line 20 of Figure 2 shows how this rule may be used to extend a dependency chain into a local variable.
3. If any pointer value that is part of a dependency chain is stored to a non-shared variable, then any value loaded by a later load from that same variable by that same thread is also part of the dependency chain. Lines 20 and 24 of Figure 2 illustrate this rule, though this rule would apply even if local variable `lsp` was a `intptr_t` instead of a pointer. Note that the job of determining whether or not a given variable is non-shared falls to the developer, *not* the implementation. That said, a high-QoI implementation might choose to make this determination in order to issue helpful diagnostic messages.
4. If a pointer value that is part of a dependency chain is stored to any variable, then any pointer value loaded by a later load from that same variable by that same thread is also part of the dependency chain. Lines 20 and 24 of Figure 2 illustrate this rule, though this rule would apply even if local variable `lsp` was instead a shared variable. As before, determining whether or not the store and load were carried out by the same thread falls to the developer, not to the implementation.
5. If a pointer value is part of a dependency chain, then casting it (either explicitly or implicitly) to any pointer type extends the chain to the result. Such casts are used heavily in the Linux kernel, for example, in the `list_for_each_entry_`

`rcu()` and `list_entry_rcu()` RCU-protected list-traversal C-preprocessor macros.

6. If a pointer value is part of a dependency chain, then if that pointer is used as a pointer-type argument of a function call, the dependency chain extends to the corresponding parameter.

7. If a function returns a pointer value that is part of a dependency chain, the dependency chain extends to the returned value in the calling function.

Pointer Offsets: If a given pointer is part of a dependency chain, then integral offsets to that pointer are also part of that dependency chain.

1. If a pointer is part of a dependency chain, then adding an integral value to that pointer extends the chain to the resulting value. This applies for both positive and negative integers, and also to addition via the infix `+` operator and via the postfix `[]` operator. Note that the addition must be carried out on a pointer: Casting to an integral type and then carrying out the addition is permitted to break the dependency chain. Therefore, instead of casting to an integral type to carry out the addition, cast to a pointer to `char`.[6] Line 24 of Figure 2 illustrates this, given that the `->t` acts as a pointer offset prior to indirection.

2. If a pointer is part of a dependency chain, then subtracting an integer from that pointer extends the chain to the resulting value. This applies for both positive and negative integers. Again, casting to an integral type and then carrying out the subtraction will break the dependency chain, so instead cast to a pointer to `char`. The Linux-kernel `container_of()` macro illustrates this. This macro is used to find the beginning of a structure given a pointer to a field within that same structure.

[6] Yes, some old systems had strange formats for character pointers, and this restriction does exclude those systems from this nuance of dependency ordering. However, to the best of my knowledge, all such systems were uniprocessors, so this is not a real problem.

```
 1 #define rcu_dereference(x) \
 2   atomic_load_explicit((x), memory_order_consume);
 3
 4 struct liststackhead {
 5   struct liststack *first;
 6 };
 7
 8 struct liststack {
 9   struct liststack *next;
10   void *t;
11   struct rcu_head rh;
12 };
13
14 void *ls_front(struct liststackhead *head)
15 {
16   void *data;
17   struct liststack *lsp;
18
19   rcu_read_lock();
20   lsp = rcu_dereference(head->first);
21   if (lsp == NULL)
22     data = NULL;
23   else
24     data = rcu_dereference(lsp->t);
25   rcu_read_unlock();
26   return data;
27 }
```

Figure 2: List-Based-Stack Whole-Program Approach, 1 of 2

3. Note that class-member access operators (`.` and `->`) can be thought of as computing an offset as part of their execution.

Dereferencing and Address-Of: Dereferencing a pointer that is part of a dependency chain extends the dependency chain to the result, but only when the resulting value is a pointer type. Taking the address of a pointer that is part of a dependency chain, and then dereferencing the resulting pointer, extends the dependency chain to the result.

1. If a pointer is part of a dependency chain, then dereferencing it using the prefix `*` operator extends the chain through the dereference operation. Line 24 of Figure 2 illustrates this, given that the `->t` acts as a pointer offset prior to indirection.

2. If a pointer is part of a dependency chain, then dereferencing it using the `->` field-selection operator extends the chain to the field. Note that the when the `->` operator is followed by one or more

```
 1 int ls_push(struct liststackhead *head, void *t)
 2 {
 3   struct liststack *lsp;
 4   struct liststack *lsnp1;
 5   struct liststack *lsnp2;
 6   size_t sz;
 7
 8   sz = sizeof(*lsp);
 9   sz = (sz + CACHE_LINE_SIZE - 1) / CACHE_LINE_SIZE;
10   sz *= CACHE_LINE_SIZE;
11   lsp = malloc(sz);
12   if (!lsp)
13     return -ENOMEM;
14   if (!t)
15     abort();
16   lsp->t = t;
17   rcu_read_lock();
18   lsnp2 = ACCESS_ONCE(head->first);
19   do {
20     lsnp1 = lsnp2;
21     lsp->next = lsnp1;
22     lsnp2 = cmpxchg(&head->first, lsnp1, lsp);
23   } while (lsnp1 != lsnp2);
24   rcu_read_unlock();
25   return 0;
26 }
27
28 static void ls_rcu_free_cb(struct rcu_head *rhp)
29 {
30   struct liststack *lsp;
31
32   lsp = container_of(rhp, struct liststack, rh);
33   free(lsp);
34 }
35
36 void *ls_pop(struct liststackhead *head)
37 {
38   struct liststack *lsp;
39   struct liststack *lsnp1;
40   struct liststack *lsnp2;
41   void *data;
42
43   rcu_read_lock();
44   lsnp2 = rcu_dereference(head->first);
45   do {
46     lsnp1 = lsnp2;
47     if (lsnp1 == NULL) {
48       rcu_read_unlock();
49       return NULL;
50     }
51     lsp = rcu_dereference(lsnp1->next);
52     lsnp2 = cmpxchg(&head->first, lsnp1, lsp);
53   } while (lsnp1 != lsnp2);
54   data = rcu_dereference(lsnp2->t);
55   rcu_read_unlock();
56   call_rcu(&lsnp2->rh, ls_rcu_free_cb);
57   return data;
58 }
```

Figure 3: List-Based-Stack Whole-Program Approach, 2 of 2

`.` operators, these latter operators are equivalent to adding a constant integer to the original pointer. Line 24 of Figure 2 directly illustrates this rule.

3. If a pointer is part of a dependency chain, then applying the unary `&` address-of operator, optionally casting this address to a pointer type (perhaps repeatedly to different pointer types, either explicitly or implicitly), then applying the `*` dereference operator extends the chain to the result. This is used by some of the Linux-kernel list-processing macros.

Miscellaneous Operations: The following operations also extend dependency chains.

1. If a pointer is part of a dependency chain, and that pointer appears in the operand of a `?:` operator selected by the condition, then the chain extends to the result. Please note that `?:` does not extend chains from its condition, only from its second or third argument.

2. If a pointer is part of a dependency chain, and that pointer appears in the right-hand operand of a `,` operator, then the chain extends to the result. Please note that the `,` operator does not extend chains from its left-hand operand, only from its right-hand operand.

3. If a pointer to a function is part of a dependency chain, then invoking the pointed-to function extends the chain from the pointer to the instructions executed. Note that the exact mechanism used to update instructions is implementation defined, and might require use of special instruction-cache-flush operations.[7]

4. If a given operation extends a dependency chain, then so does its atomic counterpart. For example, the rules applying to assignments also apply to atomic loads and stores. It also applies to

[7] This may sound strange, but just you try implementing dynamic linking without the ability to update instructions! That said, this is very clearly outside of the current standard, so this item is something to be negotiated with individual implementers.

```
 1 struct bar {
 2   struct bar *next;
 3   int a;
 4   int b;
 5 };
 6 struct bar *head = { &head, 1, 2 };
 7
 8 for (p = head->next; p; p = rcu_dereference(p->next)) {
 9   foo += p->a;
10   if (p == &head)
11     break;
12 }
13 bar *= head->b;
```

Figure 4: Back-Propagation of Dependency-Chain Breakage Due to Comparisons

```
1   if (p > &foo)
2     do_something(p);
3   else if (p < &foo)
4     do_something_else(p);
5   else
6     do_something_nodep(p);
```

Figure 5: Inequality-Comparison Dependency-Chain Breakage

atomic exchange and and atomic compare and swap.

2.2.2 Terminating Dependency Chains

Even though all other operations terminate dependency chains, there are a few that deserve special mention:

1. Equality comparisons.
2. Narrowing magnitude comparisons.
3. Narrowing arithmetic operations.
4. Narrowing bitwise operations.
5. Passing values between threads without using a `memory_order_consume` load.
6. Undefined behavior.
7. Use of `std::kill_dependency`.

Each of these is covered below.

Equality comparisons: If a pointer is part of a dependency chain, then a `==` or `!=` comparison that compares equal to some other pointer, where that other pointer is not part of any dependency chain, will cause any uses of the original pointer to no longer be part of the dependency chain. This dependency-chain breakage can back-propagate to earlier uses of the pointer, so that in Figure 4, if the comparison on line 10 compares equal, then the access on line 9 is not part of the dependency chain. This is admittedly a rather strange code fragment, and besides, the Linux-kernel `barrier()` macro could prevent this if placed between lines 9 and 10. Furthermore, the Linux kernel's list macros avoid this situation because the equal comparison terminates the loop.

So what if the implementation introduces an equality comparison? This might happen when doing feedback-directed optimization, where the implementation might notice (for example) that a particularly statically allocated structure was almost always the first element on a given list. The implementation might therefore introduce a specialization optimization, comparing the addresses and generating code using the statically allocated structure on equals comparison. On the one hand, in the cases where the Linux kernel adds a statically allocated structure to an RCU-protected linked data structure, that structure has been initialized at compile time, so that dependency ordering is not required. On the other hand, this appears to be an extremely dubious optimization for linked data structures: In a great many cases, the added overhead of the comparison would overwhelm the benefits of generating code based on the statically allocated structure.

High-quality implementations would therefore be expected to provide means for disabling this sort of optimization, especially for pointers obtained from the heap. After all, use of statically allocated structures in RCU-protected lists could be quite useful during out-of-memory conditions, in which case the specialization optimization would almost always reduce performance, which is not what optmizations are supposed to be doing.

It is tempting to insist that implementations preserve dependency chains even in the face of equality comparisons. However, such insistance elimi-

nates the possibility of a solution that does not require heavy use of explicit marking and `std::kill_dependency()`. To see this, recall that if a member of a dependency chain is stored into any variable (be it on the stack, on the heap, wherever), and if that same value is reloaded by that same thread, the dependency chain must be preserved. Allowing successful equality comparisons to break dependency chains is therefore essential to an unmarked solution to the dependency-ordering problem.

Alternatively, an intrinsic could provide comparison, but avoid breaking dependency chains. For example, a `bool std::pointer_cmp_eq_dep(T *pd, T* p)` intrinsic could compare the two pointers, but perserve dependencies carried by `pd` even if they compare equal. For completeness, a `bool std::pointer_cmp_ne_dep(T *pd, T* p)` instrinic could also be provided for not-equal comparisons. Note that there is no way to compare two pointers carrying dependencies and preserve the dependency for both. This is because there are currentely no use cases requiring this, and requiring it would require the implementation to be less efficient in its register usage.

Narrowing magnitude comparisons: A series of `>`, `<`, `>=`, or `<=` operators that informs the implementation of the exact value of a pointer causes that pointer to no longer be part of the dependency chain. See Figure 5 for an example of this. On line 6 of this figure, the implementation knows that the value of `p` is equal to `&foo`, so although there is dependency ordering to lines 2 and 4, there is no dependency ordering to line 6. This dependency-chain breakage can back-propagate, just as for equality comparisons. However, dependencies are maintained for normal uses, for example, the use of comparisons for deadlock avoidance when acquiring locks contained in multiple RCU-protected data elements.

As with equality comparison, an intrinsic could provide comparison, but avoid breaking dependency chains. For example, `bool std::pointer_cmp_gt_dep(T *pd, T* p)`, `bool std::pointer_cmp_ge_dep(T *pd, T* p)`, `bool std::pointer_cmp_lt_dep(T *pd, T* p)`, and `bool std::pointer_cmp_le_dep(T *pd, T* p)` intrinsics could compare the two pointers, but perserve dependencies carried by `pd` even if a series of comparisons allowed the implementation to deduce the exact value of the pointer.

Narrowing arithmetic operations: If a pointer is part of a dependency chain, and if the values added to or subtracted from that pointer cancel the pointer value so as to allow the implementation to precisely determine the resulting value, then the resulting value will not be part of any dependency chain. For example, if `p` is part of a dependency chain, then `((char *)p-(uintptr_t)p)+65536` will not be.[8]

Passing values between threads: If a value that is part of a dependency chain is stored into a variable by one thread, and loaded from that same variable by some other thread using either a non-atomic load or a `memory_order_relaxed` load, then the dependency chain does not extend to the second thread. To get this effect, the second thread would instead need to use a `memory_order_consume` load. Note that in theory, this would extend the dependency chain even if the corresponding store was a `memory_order_relaxed` store because the required store-side ordering is provided by the dependency chain, however, in practice the C++ standard does not guarantee ordering unless the store also has release semantics. Alternatively, the store might be preceded by an invocation of `atomic_thread_fence()` that provides release semantics.

Undefined behavior: If undefined behavior is invoked, then, consistent with the notion of undefined behavior, there are no dependency-chain guarantees.

If a given pointer takes on only one value avoids undefined behavior, then the dependency chain is broken in the same way as it would be in the case of an equality comparison with that same value. This can

[8] That said, 5.7p4 of C++ and 6.5.6p8 of C both say that indexing outside of an object is undefined behavior, so the loss of dependency ordering is likely the least of the problems here.

```
 1 int *y;
 2 int x[1];
 3
 4 int foo(void)
 5 {
 6   int *p;
 7   int t;
 8
 9   p = atomic_load_explicit(&y, memory_order_consume);
10   t = p - x;
11   return *p;
12 }
```

Figure 6: Undefined Behavior Breaks Dependency Chain

result in counter-intuitive dependency-chain breakage, as shown in Figure 6. Here, the pointer subtraction on line 10 results in undefined behavior unless `y` references some element of `x`. But since there is only one element of `x`, undefined behavior is avoided only if `y == &x[0]`, which means that the implementation knows the exact value of `y` for any valid execution. The implementation is therefore within its rights to replace line 11 with `return x[0]`, thus breaking the dependency chain.

The moral of this story is "Don't let singleton arrays anywhere near a value that carries a dependency."

kill_dependency(): The result of calling `std::kill_dependency` is never part of any dependency chain. This operation can be used to suppress diagnostics that implementations might omit for likely misuses of dependency ordering.

3 Draft Wording for Restricted Dependency Chains

This proposal recommends changes to 1.10p11 (Section 3.1), changes to 1.10p12 (Section 3.2), and, optionally, a new 29.9 (Section 3.3).

3.1 Wording for 1.10p11

1.10p11 of the most recent draft of the standard [25] is replaced with the following:

An evaluation of an expression A of pointer type *carries a dependency* to an evaluation B if

1. B is a class-member access expression (5.2.5 [expr.ref]) and A is its object expression, or
2. B is an increment or decrement expression (5.2.6 [expr.post.incr] and 5.3.2 [expr.pre.incr]) and A is its operand, or
3. B is the unary `*` dereference expression (5.3.1 [expr.unary.op]) and A is its operand, or
4. B is a standard conversion (Clause 4), a static cast (5.2.9 [expr.static.cast]), or a const cast (5.2.11 [expr.const.cast]) and A is the operand for the conversion [Note: An explicit type conversion (5.2.3 [expr.type.conv], 5.4 [expr.cast]) is analyzed in its decomposed form.], or
5. B is an additive expression (5.7 [expr.add]), A is one of its operands, and the other operand is of integral type.
6. B is the conditional expression (5.16 [expr.cond]) and A is the operand selected by its condition, or
7. B is an assignment expression (5.18 [expr.ass]) and A is its right-hand operand, or
8. B is an `+=` or `-=` compound assignment expression (5.18 [expr.ass]) and A is its right-hand operand, or
9. B is the comma operator (5.19 [expr.comma]) and A is its rightmost operand, or
10. B is an initialization (8.5 [dcl.init]) of a pointer and A is the initializer, or
11. A writes an object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or

12. for some evaluations X and Y, X is the unary `&` operator (5.3.1 [expr.unary.op]), A is that operator's operand, X carries a dependency to Y, B is the unary `*` operator (5.3.1 [expr.unary.op]), and Y is that operator's operand, or
13. for some evaluation X, A carries a dependency to X, and X carries a dependency to B.

An evaluation A carries a dependency to a side-effect C if

1. C is the side effect resulting from an increment or decrement expression (5.2.6 [expr.post.incr] and 5.3.2 [expr.pre.incr]) and A is its operand, or
2. C is the side effect resulting from the prefix dereference expression (5.3.1 [expr.unary.op]) and A is its operand, or
3. C is the side effect resulting from an assignment expression (5.18 [expr.ass]) (whether compound or not), and A is its right-hand operand.

Given an atomic type T, an evaluation A carries a dependency to a side-effect C if

1. C is the side effect resulting from a `atomic_store()` or `T::store()` function (29.6.5 [atomics.types.operations.req]), and A is the function's `object` or `desired` argument, or
2. C is the side effect resulting from an `atomic_exchange()` or `T::exchange()` function (29.6.5 [atomics.types.operations.req]), and A is the function's `object` or `desired` argument, or
3. C is the side effect resulting from an `atomic_compare_exchange_weak()`, `atomic_compare_exchange_strong()`, `T::compare_exchange_weak()`, or `T::compare_compare_exchange_strong()`, function (29.6.5 [atomics.types.operations.req]), and A is the function's `object`, `expected`, or `desired` argument or
4. C is the side effect resulting from an `atomic_fetch_add()`, `atomic_fetch_or()`, `atomic_fetch_and()`, `atomic_fetch_sub()`, `atomic_fetch_xor()`, `T::fetch_add()`, `T::fetch_or()`, `T::fetch_and()`, `T::fetch_sub()`, or `T::fetch_xor()` function (29.6.p27 [atomics.types.operations.req]), and A is either the `object` or the `operand`.

[*Note:* Carrying a dependency to a side effect ensures that the side effect is properly ordered, but does not necessarily mean that the value stored on the part of that side effect is part of the dependency chain.]

[*Note:* Carrying a dependency to a higher-level synchronization primitive, for example, mutex, lock, or call-once object (30.4 [thread.mutex]), is evaluated by looking at that primitive's implementation. However, it is expected that carrying a dependency to an object implementing such a primitive will order that primitive's operations.]

An evaluation A carries a dependency to the value stored by a side-effect C if

1. C is the value stored by the side effect resulting from an assignment expression (5.18 [expr.ass]), and A is its right-hand operand, or
2. C is the value stored by the side effect resulting from a `+=` or `-=` compound assignment expression (5.18 [expr.ass]), and A is its right-hand operand.

Given an atomic type T, an evaluation A carries a dependency to the value stored by a side-effect C if

1. C is the value stored by the side effect resulting from an `atomic_store()` or `T::store()` function (29.6.5 [atomics.types.operations.req]), and A is the function's `desired` argument, or
2. C is the value stored by the side effect resulting from an `atomic_exchange()` or `T::exchange()` function (29.6.5 [atomics.types.operations.req]), from a and A is the function's `desired` argument, or
3. C is the value stored by the side effect resulting from an `atomic_compare_exchange_weak()`, `atomic_compare_exchange_strong()`, `T::compare_exchange_weak()`, or `T::compare_compare_exchange_strong()`, function (29.6.5 [atomics.types.operations.req]), and A is the function's `desired` argument, or
4. C is the value stored by the side effect resulting from an `atomic_fetch_add()`, `atomic_fetch_or()`, `atomic_fetch_and()`, `atomic_fetch_sub()`, `atomic_fetch_xor()`, `T::fetch_add()`, `T::fetch_or()`, `T::fetch_and()`, `T::fetch_sub()`, or `T::fetch_xor()` function (29.6.5 [atomics.types.operations.req]), and A is the `operand`.

If the value C is loaded by an evaluation E sequenced after C, then a dependency is carried to E.

An evaluation A carries a dependency to a memory load D if A is sequenced before D and if

1. D is the memory load resulting from an lvalue-to-rvalue conversion and A is its glvalue expression (4.1 [conv.lval]), or
2. D is the memory load resulting from an increment or decrement expression (5.2.6 [expr.post.incr] and 5.3.2 [expr.pre.incr]) and A is its operand.

Given an atomic type T, an evaluation A carries a dependency to a memory load D if A is sequenced before D and if

1. D is the memory load resulting from an `atomic_load()` or `T::load()` function (29.6.5 [atomics.types.operations.req]), and A is the function's first argument, or
2. D is the memory load resulting from an `atomic_compare_exchange_weak()`, `atomic_compare_exchange_strong()`, `T::compare_exchange_weak()`, or `T::compare_compare_exchange_strong()`, function (29.6.5 [atomics.types.operations.req]), and A is the function's `object` argument, or
3. D is the memory load resulting from an `atomic_exchange()` or `T::exchange()` function (29.6.5 [atomics.types.operations.req]), and A is the function's `object` argument, or
4. D is the memory load resulting from an `atomic_fetch_add()`, `atomic_fetch_or()`, `atomic_fetch_and()`, `atomic_fetch_sub()`, `atomic_fetch_xor()`, `T::fetch_add()`, `T::fetch_or()`, `T::fetch_and()`, `T::fetch_sub()`, or `T::fetch_xor()` function (29.6.p27 [atomics.types.operations.req]), and A is the function's `object` argument.

[*Note:* Carrying a dependency to a memory load ensures that the load is properly ordered, but does not necessarily mean that the value loaded is part of the dependency chain.]

If the implementation is able to exactly determine the value of a pointer, for example, due to undefined-behavior analysis or due to a series of comparison operators that enables exact determination of its value, then no dependency is carried to that pointer.

[*Note:* To carry out comparisons without destroying the depen-

dency chain, use the `std::pointer_cmp_eq_dep()`, `std::pointer_cmp_ne_dep()`, `std::pointer_cmp_gt_dep()`, `std::pointer_cmp_ge_dep()`, `std::pointer_cmp_lt_dep()`, or `std::pointer_cmp_le_dep()` intrinsics if they are provided.]

[*Note:* `std::kill_dependency()` is not mentioned above, so if A is its argument and B its result, no dependency is carried from A to B.]

[*Note:* The intent of these rules is to enable implementations to carry out their normal optimizations, while still permitting developers to rely on the common dependency-ordering use cases.]

[*Note:* Although the `[[carries_dependency]]` attribute[9] is no longer needed to specify that a dependency chain exists, this attribute can help the implementation provide higher-quality diagnostics. For example, doing an equality comparison to a non-`NULL` pointer that is dereferenced could result in a warning diagnostic. Such a warning would make the user aware that no dependency was carried to that pointer.]

3.2 Wording for 1.10p12

1.10p12 of the most recent draft is updated by inserting the phrase "of pointer type", resulting in'the following:

> An evaluation A is *dependency-ordered before* an evaluation B if
>
> - A performs a release operation on an atomic object M of pointer type, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
> - for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[9] Some spelling other than `[[carries_dependency]]` might well be chosen at some point.

```
 1 struct rcutest {
 2   int a;
 3   int b;
 4   int c;
 5 };
 6
 7 struct rcutest1 {
 8   int a;
 9   struct rcutest rt;
10 };
11
12 struct rcutest *gp;
13 struct rcutest *gslp; /* Global scope, local usage. */
14
15 #define rcu_assign_pointer(p, v) \
16   atomic_store_explicit(&(p), (v), memory_order_release);
17 #define rcu_dereference_pointer(p) \
18   atomic_load_explicit(&(p), memory_order_consume);
```

Figure 7: Litmus-Test Definitions

3.3 Optional Wording for 29.9

A new section 29.9 could add intrinsics for dependency-preserving comparisons:

```
extern "C" bool pointer_cmp_eq_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_ne_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_gt_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_ge_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_lt_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_le_dep(T *pd, T *p) noexcept;
```

> *Effects:* carries out the specified comparison of the two pointers while preserving any dependencies carried through **pd**. Note that dependencies carried through **p** may be lost.

An alternative approach would be to rely on variable marking to get the same effect, thus dispensing with the above intrinsics.

4 Litmus Tests

Figure 7 shows some common definitions used by multiple litmus tests, each of which is discussed in one of the following sections.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   assert(p->a != 43);
 9   rcu_assign_pointer(gp, p);
10 }
11
12 void thread1(void)
13 {
14   struct rcutest *p;
15
16   p = rcu_dereference(gp);
17   if (p)
18     p->a = 43;
19 }
```

Figure 8: Litmus Test: Simple Left-Hand-Side Dependency

4.1 Simple Left-Hand-Side Dependency

Figure 8 shows a simple left-hand-side dependency. Dependency ordering guarantees that the assertion on line 8 never triggers.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14
15   p = rcu_dereference(gp);
16   if (p)
17     assert(p->a == 42);
18 }
```

Figure 9: Litmus Test: Simple Right-Hand-Side Dependency

4.2 Simple Right-Hand-Side Dependency

Figure 9 shows a simple right-hand-side dependency. Dependency ordering guarantees that the assertion on line 17 never triggers.

4.3 Local Storage For Dependency

Figure 9 also demonstrates that storing a pointer to a local variable preserves the dependency chain. The pointer stored to local variable `p` on line 15 is part of a dependency chain, and the dependency chain remains when it is reloaded on line 17. Dependency ordering thus continues to guarantee that the assertion on line 8 never triggers.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14
15   p = rcu_dereference(gp);
16   gslp = p;
17   p = gslp;
18   if (p)
19     assert(p->a == 42);
20 }
```

Figure 10: Litmus Test: Non-Local Storage For Dependency

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14
15   p = rcu_dereference(gp);
16   gslp = p;
17 }
18
19 void thread2(void)
20 {
21   struct rcutest *p;
22
23   p = gslp;
24   if (p)
25     assert(p->a == 42);
26 }
```

Figure 11: Litmus Test: Non-Local Storage and Reload Kills Dependency

4.4 Non-Local Storage For Dependency

Figure 10 shows that storing to a non-local variable preserves a dependency if this value is later reloaded by the same thread. Dependency ordering guarantees that the assertion on line 19 never triggers, despite the store to and reload from `gslp`.

Note that this pattern must be used with care because dependency ordering is *not* guaranteed when the reload is done by some other thread, as shown by the next litmus test.

4.5 Non-Local Storage and Reload Kills Dependency

Figure 11 shows that storing to a non-local variable, then reloading from that variable within some other thread, kills the dependency. Dependency ordering therefore does nothing to prevent the assertion on line 25 from triggering.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14   struct rcutest1 *q;
15
16   p = rcu_dereference(gp);
17   q = (struct rcutest1)p;
18   if (q)
19     assert(q->a == 42);
20 }
```

Figure 12: Litmus Test: Casting For Dependency

4.6 Casting For Dependency

Figure 12 shows that casting a pointer to another pointer type with compatible layout preserves dependency ordering so that the assertion on line 19 never triggers, despite the cast operation.

4.7 Casting to Non-Pointer Kills Dependency

Figure 13 shows that casting a pointer to a non-pointer type can kill the dependency, despite the later cast back to a pointer type. The assertion on line 20 can therefore trigger.

4.8 Function Argument Carries Dependency

Figure 14 shows that passing a pointer via a function argument preserves dependency ordering so that the assertion on line 14 never triggers. As shown in Figure 15, the `[[carries_dependency]]` attribute could be used to document the fact that the developer expects a dependency chain to pass into the function via this parameter, and implementations might use these attributes to improve diagnostics.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14   uintptr_t q;
15
16   p = rcu_dereference(gp);
17   q = (uintptr_t)p;
18   p = (struct rcutest *)q;
19   if (p)
20     assert(p->a == 42);
21 }
```

Figure 13: Litmus Test: Casting to Non-Pointer Kills Dependency

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1_help(struct rcutest *q)
12 {
13   if (q)
14     assert(q->a == 42);
15 }
16
17 void thread1(void)
18 {
19   struct rcutest *p;
20
21   p = rcu_dereference(gp);
22   thread1_help(p);
23 }
```

Figure 14: Litmus Test: Function Argument Carries Dependency

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1_help(struct rcutest *q [[carries_dependency]])
12 {
13   if (q)
14     assert(q->a == 42);
15 }
16
17 void thread1(void)
18 {
19   struct rcutest *p;
20
21   p = rcu_dereference(gp);
22   thread1_help(p);
23 }
```

Figure 15: Litmus Test: Function Argument Explicitly Carries Dependency

4.9 Function Return Carries Dependency

Figure 14 shows that returning a pointer from a function preserves dependency ordering so that the assertion on line 22 never triggers. As shown in Figure 17, the `[[carries_dependency]]` attribute can again be used to make the dependency-carrying explicit.

4.10 Array-Offset Dependency

Figure 18 shows how dependencies survive addition of integer offsets via array indexing. Dependency ordering guarantees that the assertion on line 18 never triggers.

4.11 Integer-Pointer Addition Dependency

Figure 19 shows how dependencies survive direct addition of integer offsets. Dependency ordering guarantees that the assertion on line 18 never triggers.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 struct rcutest *thread1_help(void)
12 {
13   return rcu_dereference(gp);
14 }
15
16 void thread1(void)
17 {
18   struct rcutest *p;
19
20   p = thread1_help();
21   if (q)
22     assert(q->a == 42);
23 }
```

Figure 16: Litmus Test: Function Return Carries Dependency

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 [[carries_dependency]] struct rcutest *thread1_help(void)
12 {
13   return rcu_dereference(gp);
14 }
15
16 void thread1(void)
17 {
18   struct rcutest *p;
19
20   p = thread1_help();
21   if (q)
22     assert(q->a == 42);
23 }
```

Figure 17: Litmus Test: Function Return Explicitly Carries Dependency

```
 1 void thread0(void)
 2 {
 3   int *p;
 4
 5   p = malloc(4 * sizeof(*p));
 6   assert(p);
 7   p[0] = 1;
 8   p[1] = 42;
 9   rcu_assign_pointer(gp, p);
10 }
11
12 void thread1(void)
13 {
14   struct rcutest *p;
15
16   p = rcu_dereference(gp);
17   if (p)
18     assert(p[p[0]] == 42);
19 }
```

Figure 18: Litmus Test: Array-Offset Dependency

```
 1 void thread0(void)
 2 {
 3   int *p;
 4
 5   p = malloc(4 * sizeof(*p));
 6   assert(p);
 7   p[0] = 1;
 8   p[1] = 42;
 9   rcu_assign_pointer(gp, p);
10 }
11
12 void thread1(void)
13 {
14   struct rcutest *p;
15
16   p = rcu_dereference(gp);
17   if (p)
18     assert(*(p + p[0]) == 42);
19 }
```

Figure 19: Litmus Test: Integer-Pointer Addition Dependency

```
 1 void thread0(void)
 2 {
 3   int *p;
 4
 5   p = malloc(4 * sizeof(*p));
 6   assert(p);
 7   p[1] = -1;
 8   p[0] = 42;
 9   rcu_assign_pointer(gp, &p[1]);
10 }
11
12 void thread1(void)
13 {
14   struct rcutest *p;
15
16   p = rcu_dereference(gp);
17   if (p)
18     assert(*(p + p[0]) == 42);
19 }
```

Figure 20: Litmus Test: Integer-Pointer Subtraction Dependency

4.12 Integer-Pointer Subtraction Dependency

Figure 20 shows how dependencies survive direct subtraction of integer offsets. Dependency ordering guarantees that the assertion on line 18 never triggers.

4.13 Field-Selection Offset Dependency

Figure 21 shows that dependencies are carried through offsets due to the field-selection operator. Dependency ordering guarantees that the assertion on line 17 never triggers.

4.14 Direct Dereferencing Dependency

Figure 22 shows how dependencies direct dereferencing. Dependency ordering guarantees that the assertion on line 17 never triggers.

4.15 Enclosing-Structure Location Dependency

Figure 23 shows that dependencies are carried through offsets used to locate an enclosing structure, an idiom used by the `container_of()` macro in the

```
 1 void thread0(void)
 2 {
 3   struct rcutest1 *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->rt.a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14
15   p = rcu_dereference(gp);
16   if (p)
17     assert(p->rt.a == 42);
18 }
```

Figure 21: Litmus Test: Field-Selection Offset Dependency

```
 1 void thread0(void)
 2 {
 3   int *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   *p = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void thread1(void)
12 {
13   struct rcutest *p;
14
15   p = rcu_dereference(gp);
16   if (p)
17     assert(*p == 42);
18 }
```

Figure 22: Litmus Test: Direct Dereferencing Dependency

```
 1 void thread0(void)
 2 {
 3   struct rcutest1 *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, &p->rt);
 9 }
10
11 void thread1(void)
12 {
13   char *cp;
14   struct rcutest *p;
15   struct rcutest1 *q;
16
17   p = rcu_dereference(gp);
18   if (p) {
19     cp = (char *)p;
20     cp -= (uintptr_t)&((struct rcutest1 *)NULL)->rt;
21     q = (struct rcutest1)cp;
22     assert(q->a == 42);
23   }
24 }
```

Figure 23: Litmus Test: Enclosing-Structure Location Dependency

Linux kernel. Dependency ordering guarantees that the assertion on line 22 never triggers.

4.16 Conditional-Expression Dependency

Figure 24 shows a dependency carried through a conditional expression. Dependency ordering guarantees that the assertion on line 20 never triggers.

4.17 Comma-Expression Dependency

Figure 25 shows a dependency carried through the right-hand operand of a comma expression Dependency ordering guarantees that the assertion on line 20 never triggers.

4.18 Equality Comparisons Kill Dependency

Figure 26 shows a dependency chain being killed by the equality comparison on line 19. The implementation is within its rights to substitute `i = rt.b` for line 20, which the implementation can hoist to precede line 19, even on a strongly ordered system. This

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   p->b = 43;
 9   rcu_assign_pointer(gp, p);
10 }
11
12 void thread1(void)
13 {
14   int i;
15   struct rcutest *p;
16
17   p = rcu_dereference(gp);
18   if (p) {
19     i = (random() & 0x1) ? p->a : p->b;
20     assert(i == 42 || i == 43);
21   }
22 }
```

Figure 24: Litmus Test: Conditional-Expression Dependency

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = malloc(sizeof(*p));
 6   assert(p);
 7   p->a = 42;
 8   p->b = 43;
 9   rcu_assign_pointer(gp, p);
10 }
11
12 void thread1(void)
13 {
14   int i;
15   struct rcutest *p;
16
17   p = rcu_dereference(gp);
18   if (p) {
19     i = p->b , p->a;
20     assert(i == 42);
21   }
22 }
```

Figure 25: Litmus Test: Comma-Expression Dependency

```
 1 struct rcutest rt = { 1, 2, 3 };
 2
 3 void thread0(void)
 4 {
 5   rt.a = -42;
 6   rt.b = -43;
 7   rt.c = -44;
 8   rcu_assign_pointer(gp, &rt);
 9 }
10
11 void thread1(void)
12 {
13   int i = -1;
14   int j = -1;
15   struct rcutest *p;
16
17   p = rcu_dereference(gp);
18   j = p->a;
19   if (p == &rt)
20     i = p->b; /* Dependency chain broken! */
21   else if (p)
22     i = p->c;
23   assert(i < 0);
24   assert(j < 0);
25 }
```

Figure 26: Litmus Test: Equality Comparisons Kill Dependency

hoisting allows access to pre-initialization values for `rt.b` which would be the constant `2` from the initializer on line 1. In this case, the assertion on line 23 would trigger.

Worse yet, the implemenation would be within its rights to transform the code as shown in Figure 27. In this case, both assertions can clearly trigger. In other words, an equality comparison can break dependencies in code *preceding* the dependency.

4.19 Equality Comparisons Without Killing Dependency

Figure 28 shows a how `pointer_cmp_eq_dep()` and friends provide a way of carrying out comparisons without killing dependency chains. With the dependency chain preserved, the assertions on lines 23 and 24cannot trigger.

```
 1 struct rcutest rt = { 1, 2, 3 };
 2
 3 void thread0(void)
 4 {
 5   rt.a = -42;
 6   rt.b = -43;
 7   rt.c = -44;
 8   rcu_assign_pointer(gp, &rt);
 9 }
10
11 void thread1(void)
12 {
13   int i = -1;
14   int j = -1;
15   struct rcutest *p;
16
17   i = rt.b;
18   j = rt.a;
19   p = rcu_dereference(gp);
20   if (p && p != &rt) {
21     i = p->c;
22     j = p->a;
23   } else if (!p) {
24     i = j = -1;
25   }
26   assert(i < 0);
27   assert(j < 0);
28 }
```

Figure 27: Litmus Test: Equality Comparisons Kill Dependency, Optimized

```
 1 struct rcutest rt = { 1, 2, 3 };
 2
 3 void thread0(void)
 4 {
 5   rt.a = -42;
 6   rt.b = -43;
 7   rt.c = -44;
 8   rcu_assign_pointer(gp, &rt);
 9 }
10
11 void thread1(void)
12 {
13   int i = -1;
14   int j = -1;
15   struct rcutest *p;
16
17   p = rcu_dereference(gp);
18   j = p->a;
19   if (pointer_cmp_eq_dep(p, &rt))
20     i = p->b;
21   else if (p)
22     i = p->c;
23   assert(i < 0);
24   assert(j < 0);
25 }
```

Figure 28: Litmus Test: Equality Comparisons Without Killing Dependency

5 Benefits, Drawbacks, and Mitigations

This section compares the proposal to the requirements and desiderata called out in Section 2.1.

As required, dependency chains do not depend solely on marking objects carrying dependencies. Such objects may be marked, if desired, using `[[carries_dependency]]`, which can improve diagnostics and perhaps also improve formal verification.

The head of a dependency chain is marked with a `memory_order_consume` load. No other markings are needed, although `[[carries_dependency]]` may be used as noted above. In addition, `pointer_cmp_eq_dep()`, `pointer_cmp_ne_dep()`, `pointer_cmp_gt_dep()`, `pointer_cmp_ge_dep()`, `pointer_cmp_lt_dep()`, and `pointer_cmp_le_dep()`, if provided, could be used to avoid dependency-chain breakage that might otherwise occur due to value-narrowing comparisons.

Dependency chains have been defined to avoid the need for explicit memory-barrier instructions on mainstream systems.

Dependencies need be carried only through pointers and to non-pointers.

Implementations need not trace dependency chains. Again, `[[carries_dependency]]` may be used to permit improved diagnostics, but without the need to trace dependency chains.

It is important to note that this proposal does not address the need to tag pointers via low-order bits. Such tagging is required for a number of algorithms for concurrency and for memory allocation. Should a future proposal to define pointer-bit tagging be accepted, that proposal should accommodate carrying dependencies through tagged pointers.

In summary, this proposal provides the features needed by most existing RCU-related practice without placing undue burdens on the implementations.

References

[1] Arbel, M., and Morrison, A. Predicate rcu: An rcu for scalable concurrent updates. *SIGPLAN Not. 50*, 8 (Jan. 2015), 21–30.

[2] Ash, M. Concurrent memory deallocation in the objective-c runtime. mikeash.com: just this guy, you know?, May 2015.

[3] Compaq Computer Corporation. Shared memory, threads, interprocess communication. Available: `http://h71000.www7.hp.com/wizard/wiz_2637.html`, August 2001.

[4] Desnoyers, M. [RFC git tree] userspace RCU (urcu) for Linux. `http://liburcu.org`, February 2009.

[5] Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23* (2012), 375–382.

[6] Kung, H. T., and Lehman, P. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382.

[7] Liu, R., Zhang, H., and Chen, H. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 219–230.

[8] Liu, Y., Luchangco, V., and Spear, M. Mindicators: A scalable approach to quiescence. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems* (Washington, DC, USA, 2013), ICDCS '13, IEEE Computer Society, pp. 206–215.

[9] Matveev, A., Shavit, N., Felber, P., and Marlier, P. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 168–183.

[10] McKenney, P. E. What is RCU? part 2: Usage. Available: `http://lwn.net/Articles/`

263130/ [Viewed January 4, 2008], January 2008.

[11] McKenney, P. E. Transactional memory everywhere? http://paulmck.livejournal.com/tag/transactional%20memory%20everywhere, September 2009.

[12] McKenney, P. E. The RCU API, 2010 edition. http://lwn.net/Articles/418853/, December 2010.

[13] McKenney, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012.

[14] McKenney, P. E. Structured deferral: synchronization via procrastination. *Commun. ACM 56*, 7 (July 2013), 40–49.

[15] McKenney, P. E. [PATCH tip/core/rcu 1/4] mce: Stop using array-index-based RCU primitives. [PATCHtip/core/rcu1/4]mce: Stopusingarray-index-basedRCUprimitives, May 2015.

[16] McKenney, P. E., Purcell, C., Algae, Schumin, B., Cornelius, G., Qwertyus, Conway, N., Sbw, Blainster, Rufus, C., Zoicon5, Anome, and Eisen, H. Read-copy update. http://en.wikipedia.org/wiki/Read-copy-update, July 2006.

[17] McKenney, P. E., Riegel, T., Preshin, J., Boehm, H., Nelson, C., Giroux, O., and Crowl, L. Towards implementation and use of memory_order_consume. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0098r0.pdf, September 2015.

[18] McKenney, P. E., and Walpole, J. What is RCU, fundamentally? Available: http://lwn.net/Articles/262464/ [Viewed December 27, 2007], December 2007.

[19] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating systems transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 161–176.

[20] Porter, D. E., and Witchel, E. Lessons from large transactional systems. Personal communication ¡20071214220521.GA5721@olive-green.cs.utexas.edu¿, December 2007.

[21] Ramalhete, P., and Correia, A. Poor man's rcu. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/poormanurcu-2015.pdf, August 2015.

[22] Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Bhandari, A., and Witchel, E. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems Principles* (October 2007), ACM SIGOPS. Available: http://www.sosp2007.org/papers/sosp056-rossbach.pdf [Viewed October 21, 2007].

[23] Sites, R. L., and Witek, R. T. *Alpha AXP Architecture*, second ed. Digital Press, 1995.

[24] Sivaramakrishnan, K., Ziarek, L., and Jagannathan, S. Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management* (New York, NY, USA, 2012), ISMM '12, ACM, pp. 49–60.

[25] Smith, R. Working draft, standard for programming language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4527.pdf, May 2015.

Change Log

This paper was first posted informally in January of 2016. Revisions to this initial document are as follows:

- Add litmus tests in Section 4 as requested by Michael Wong. (January 8, 2016.)

- Apply feedback from Hans, including noting non-RCU use cases in the preamble starting on the first page, more clearly distinguishing this proposal from the wording in the current standard throughout, providing better justification for equality comparisons breaking dependency chains, adding Hans's litmus test showing how avoiding undefined behavior can break dependency chains in a manner similar to equality comparisons, and marking as optional the wording for 29.9 (intrinsics for dependency-preserving equality comparison). (January 12, 2016.)
- Apply wording feedback from Jens Maurer. (January 13, 2016.)