P0462R0:
Marking `memory_order_consume` Dependency Chains

Doc. No.: WG21/P0462R0
Date: 2016-10-13
Reply to: Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, Olivier Giroux, Lawrence Crowl, JF Bastien, and Micheal Wong
Email: paulmck@linux.vnet.ibm.com, triegel@redhat.com, jeff@preshing.com boehm@acm.org, clark.nelson@intel.com, OGiroux@nvidia.com, Lawrence@Crowl.org, jfbastien@apple.com, and fraggamuffin@gmail.com

Other contributors: Alec Teal, Alisdair Meredith, 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, Thomas Koeppe, Jens Maurer, ...

October 13, 2016

This document is based in part on WG21/D0098R1, extracting the alternatives for marking dependency chains (each headed by a `memory_order_consume` [1] load). It also adds a few additional alternatives based on discussions at the March 2016 meeting in Jacksonville, Florida, This document does not define the behavior of dependency chains, but instead only the syntax used to call the compiler's attention to them. Please see WG21/P0190R2 for detailed information on dependency-chain behavior.

1 Introduction

Spirited discussions of `memory_order_consume` at the Jacksonville meeting resulted in a few of items of agreement:

1. Dependency chains should be restricted to pointers. Please note that this excludes not only the troublesome objects of integral type, but also accesses to static members of classes.
2. Unmarked code can be handled by having the implementation behave as if markings had been supplied in all locations that could reasonably be marked. This allows natural handling of dependencies in unmarked code. This behavior should be controlled by a compiler flag. Such a flag is of course outside of the standard.
3. Software artifacts that are built standalone (such as the Linux kernel and numerous embedded projects) can reasonably use unmarked dependency chains. In contrast, software artifacts that are expected to dynamically link against standard libraries seem likely to need to mark their dependency chains.
4. Discussions involving marking of library APIs have been set aside for the moment, and so this

document does not address this point.

These points result in three known valid ways of handling `memory_order_consume`:

1. Ignore the markings and promote `memory_order_consume` to `memory_order_acquire`, as is current practice.
2. Ignore the markings, demote `memory_order_consume` to `memory_order_relaxed`, and suppress troublesome optimizations of pointers. However, there was some difficulty in arriving at a precise definition of "troublesome".
3. Demote `memory_order_consume` to `memory_order_relaxed` and suppress troublesome optimizations of marked pointers. The fact that such optimizations need not be suppressed for unmarked pointers means that a much more conservative definition of "troublesome" is feasible, thus reducing the need for precision. Note that pointer comparisons will still break dependency chains in some cases, unless the comparisons were carried out using proposed dependency-preserving pointer-comparison intrinsics. Note further that the template-based method described in Section 3.4.4 uses operator overloading so that the usual relational operators invoke these intrinsics.

However, a number of ways of marking dependency chains have been proposed and there was nothing resembling any sort of agreement on which should be used. This paper therefore catalogs approaches to marking dependency chains, and evaluates each against a set of representative use cases.

2 Representative Use Cases

This section uses the common definitions shown in Figure 1 to discuss the use cases in the following list:

1. Simple case.
2. Function in via parameter.
3. Function out via return value.

```
struct rcutest {
  int a;
  int b;
  int c;
  spinlock_t lock;
};

struct rcutest1 {
  int a;
  struct rcutest rt;
};

std::atomic<rcutest *> gp;
std::atomic<rcutest1 *> g1p;
std::atomic<int *> gip;
struct rcutest *gslp; /* Global scope, local usage. */
std::atomic<rcutest *> gsgp;

template<typename T>
T *rcu_consume(std::atomic<T*> *p)
{
  volatile std::atomic<T> *q = p;
  // Change to memory_order_consume once it is fixed
  depending_ptr<T> temp(q->load(std::memory_order_relaxed));
  return temp;
}

template<typename T>
T *rcu_consume(T *p)
{
  // Alternatively, could cast p to volatile atomic...
  T *temp(*(T *volatile *)&p);
  return temp;
}

template<typename T>
T* rcu_store_release(std::atomic<T*> *p, T *v)
{
  p->store(v, std::memory_order_release);
  return v;
}

template<typename T>
T* rcu_store_release(T **p, T *v)
{
  // Alternatively, could cast p to volatile atomic...
  atomic_thread_fence(std::memory_order_release);
  *((volatile T **)p) = v;
  return v;
}

// Linux-kernel compatibility macros, not for atomics
#define rcu_dereference(p) rcu_consume(p)
#define rcu_assign_pointer(p, v) rcu_store_release(&(p), v)
```

Figure 1: Common Definitions

```
 1 void *thread0(void *unused)
 2 {
 3   rcutest *p;
 4
 5   p = new rcutest();
 6   assert(p);
 7   p->a = 42;
 8   assert(p->a != 43);
 9   rcu_store_release(&gp, p);
10   return nullptr;
11 }
12
13 void *thread1(void *unused)
14 {
15   rcutest p;
16
17   p = rcu_consume(&gp);
18   if (p)
19     p->a = 43;
20   return nullptr;
21 }
```

Figure 2: Simple Case

4. Function both in and out, but different chains.
5. Dependency chain fanning out.
6. Dependency chain fanning in.
7. Dependency chain fanning both in and out.
8. Conditional compilation of endpoint accesses.
9. Examples involving handoff to locking.

Each of the above use cases is covered in one of the following sections, followed by a discussion of evaluation criteria.

2.1 Simple Case

The simple case is shown in Figure 2. Here, the dependency chain extends from line 16 through line 18, where it terminates. Given the simplicity and compactness of this example, any reasonable proposal should handle this example simply and naturally.

2.2 In via Function Parameter

Figure 3 shows an example dependency chain that begins at line 22, enters function `thread1_help()` at line 23, and then extending from line 12 to line 15 in the called function. This is a common encapsulation technique.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4
 5   p = new rcutest;
 6   assert(p);
 7   p->a = 42;
 8   rcu_assign_pointer(gp, p);
 9 }
10
11 void
12 thread1_help(struct rcutest *q)
13 {
14   if (q)
15     assert(q->a == 42);
16 }
17
18 void thread1(void)
19 {
20   struct rcutest *p;
21
22   p = rcu_dereference(gp);
23   thread1_help(p);
24 }
```

Figure 3: In via Function Parameter

2.3 Out via Function Return

Figure 4 shows a dependency chain exiting a function. It starts at line 21, is returned to line 20, and terminates on line 22. This is also a common encapsulation technique.

2.4 In and Out, But Different Chains

Figure 5 shows an example where a dependency chain enters a function (`thread1_help()` on lines 16-21) and a dependency chain leaves that same function, but where they are different chains.

2.5 Chain Fanning Out

Figure 6 shows a dependency chain fanning out, courtesy of the `thread1()` function's calls to `thread1_help1()` and `thread1_help2()` on lines 30 and 31. This is a common pattern in the Linux kernel, as it supports abstraction of data structures, for example, allowing common RCU-protected data structures to be aggregated into a larger RCU-protected data structure. In this scenario, `thread1_help1()` might implement one type of RCU-protected structure and `thread1_help2()` might implement another.

```
void thread0(void)
{
  struct rcutest *p;

  p = new rcutest;
  assert(p);
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

struct rcutest *thread1_help(void)
{
  return rcu_dereference(gp);
}

void thread1(void)
{
  struct rcutest *p;

  p = thread1_help();
  if (p)
    assert(p->a == 42);
}
```

Figure 4: Out via Function Return

```
void thread0(void)
{
  struct rcutest *p;

  p = new rcutest;
  assert(p);
  p->a = 42;
  rcu_assign_pointer(gp, p);

  p = new rcutest;
  assert(p);
  p->a = 43;
  rcu_assign_pointer(gsgp, p);
}

struct rcutest *thread1_help(struct rcutest *p)
{
  if (p)
    assert(p->a == 42);
  return rcu_dereference(gsgp);
}

void thread1(void)
{
  struct rcutest *p;

  p = rcu_dereference(gp);
  p = thread1_help(p);
  if (p)
    assert(p->a == 43);
}
```

Figure 5: In and Out, But Different Chains

```
void thread0(void)
{
  struct rcutest *p;

  p = new rcutest;
  assert(p);
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

void
thread1_help1(struct rcutest *q)
{
  if (q)
    assert(q->a == 42);
}

void
thread1_help2(struct rcutest *q)
{
  if (q)
    assert(q->a != 43);
}

void thread1(void)
{
  struct rcutest *p;

  p = rcu_dereference(gp);
  thread1_help1(p);
  thread1_help2(p);
}
```

Figure 6: Chain Fanning Out

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4   struct rcutest1 *p1;
 5
 6   p = new rcutest;
 7   assert(p);
 8   p->a = 42;
 9   rcu_assign_pointer(gp, p);
10
11   p1 = new rcutest;
12   assert(p1);
13   p1->a = 41;
14   p1->rt.a = 42;
15   rcu_assign_pointer(g1p, p1);
16 }
17
18 void
19 thread1_help(struct rcutest *q)
20 {
21   if (q)
22     assert(q->a == 42);
23 }
24
25 void thread1(void)
26 {
27   struct rcutest *p;
28
29   p = rcu_dereference(gp);
30   thread1_help(p);
31 }
32
33 void thread2(void)
34 {
35   struct rcutest1 *p1;
36
37   p1 = rcu_dereference(g1p);
38   thread1_help(&p1->rt);
39 }
```

Figure 7: Chain Fanning In

2.6 Chain Fanning In

Figure 7 demonstrates different dependency chains fanning into the same function, in this case `thread1_help()`, from lines 29 and 37. This fanning-in is also used to support abstraction, for example, allowing a given implementation of an RCU-protected data structure to be aggregated into several different data structures.

2.7 Chain Fanning In and Out

Figure 8 shows dependency chains fanning both in and out, starting at lines 45 and 53, fanning into `thread1_help()`, and fanning out again at the call to `thread1a_help()` on line 36 and to `thread1b_help()`

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4   struct rcutest1 *p1;
 5
 6   p = new rcutest;
 7   assert(p);
 8   p->a = 42;
 9   p->b = 43;
10   rcu_assign_pointer(gp, p);
11
12   p1 = new rcutest;
13   assert(p1);
14   p1->a = 41;
15   p1->rt.a = 42;
16   p1->rt.b = 43;
17   rcu_assign_pointer(g1p, p1);
18 }
19
20 void
21 thread1a_help(struct rcutest *q)
22 {
23   assert(q->a == 42);
24 }
25
26 void
27 thread1b_help(struct rcutest *q)
28 {
29   assert(q->b == 43);
30 }
31
32 void
33 thread1_help(struct rcutest *q)
34 {
35   if (q) {
36     thread1a_help(q);
37     thread1b_help(q);
38   }
39 }
40
41 void thread1(void)
42 {
43   struct rcutest *p;
44
45   p = rcu_dereference(gp);
46   thread1_help(p);
47 }
48
49 void thread2(void)
50 {
51   struct rcutest1 *p1;
52
53   p1 = rcu_dereference(g1p);
54   thread1_help(&p1->rt);
55 }
```

Figure 8: Chain Fanning In and Out

on line 37. This combination permits composition of the types of abstraction described in Sections 2.5 and 2.6.

2.8 Conditional Compilation of Chain Endpoints

Although the C preprocessor does not necessarily have the best reputation among the various aspects of either C or C++, it is true that it is always there when you need it. Figure 9 applies conditional compilation to Figure 8, so that portions of the dependency chain can come and go, depending on the value of the C-preprocessor macro `FOO`.

```
 1 void thread0(void)
 2 {
 3   struct rcutest *p;
 4   struct rcutest1 *p1;
 5
 6   p = new rcutest;
 7   assert(p);
 8   p->a = 42;
 9   p->b = 43;
10   rcu_assign_pointer(gp, p);
11
12   p1 = new rcutest;
13   assert(p1);
14   p1->a = 41;
15   p1->rt.a = 42;
16   p1->rt.b = 43;
17   rcu_assign_pointer(g1p, p1);
18 }
19
20 #ifdef FOO
21 void
22 thread1a_help(struct rcutest *q)
23 {
24   assert(q->a == 42);
25 }
26 #endif
27
28 void
29 thread1b_help(struct rcutest *q)
30 {
31   assert(q->b == 43);
32 }
33
34 void
35 thread1_help(struct rcutest *q)
36 {
37   if (q) {
38 #ifdef FOO
39     thread1a_help(q);
40 #endif
41     thread1b_help(q);
42   }
43 }
44
45 void thread1(void)
46 {
47   struct rcutest *p;
48
49   p = rcu_dereference(gp);
50   thread1_help(p);
51 }
52
53 void thread2(void)
54 {
55   struct rcutest1 *p1;
56
57   p1 = rcu_dereference(g1p);
58   thread1_help(&p1->rt);
59 }
```

Figure 9: Conditional Compilation of Chain Endpoints

2.9 Handoff to Locking

Figure 10 shows how RCU protection can hand off to other synchronization primitives, in this case, locking. The dependency chain starts at line 16 and continues through line 18 and 19. However, once line 19 has completed, the code is under the protection of `p->lock`, so line 20 explicitly ends the dependency chain. The lock then protects the increment on line 21.

It is also possible to hand off protection from RCU to reference counting, explicit memory barriers, transactional memory, and so on.

Note that the `std::kill_dependency()` on line 20 will typically have no effect on code generation.

2.10 Evaluation Criteria

1. Ease of compilation.
2. Ease of modification of programs.
3. Precise specification of dependency chains.
4. Support for cross-function dependency chains.
5. Support for cross-compilation-unit dependency chains.
6. Compatibility with C.
7. Formal Verification Compatibility.

```
void thread0(void)
{
  struct rcutest *p;

  p = new rcutest;
  assert(p);
  p->a = 42;
  assert(p->a != 43);
  rcu_assign_pointer(gp, p);
}

void thread1(void)
{
  struct rcutest *p;

  p = rcu_dereference(gp);
  if (p) {
    assert(p->a == 42);
    spin_lock(&p->lock);
    p = std::kill_dependency(p);
    p->a++;
    spin_unlock(&p->lock);
  }
}
```

Figure 10: Handoff to Locking

3 Marking Proposals

The following sections present alternative marking proposals. Whatever proposal is chosen, implementors are encouraged to provide a means (for example, a command-line argument) to cause the implementation to act as if markings were placed everywhere they could reasonably be placed. This approach permits unmarked programs containing dependency chains to be handled in a reasonably natural manner.

Note that many of these proposals consist only of short descriptions. Only proposals having proponents willing to fill them out should be considered for standardization.

3.1 Mark Translation Unit

Within the language, translation-unit marking could be accomplished by a pragma or by a language feature that changed the way pointers are implemented. A compiler command-line argument could also be used, but this is of course outside the standard. It would be desirable for marked translation units to be able to be linked with unmarked translation units.

This approach could be useful in cases where only a few of the translation units contain dependency chains. However, software-engineering considerations would likely cause many such projects to mark all the translation units, which would of course result in the same dependency-chain-tracing complexity as would unmarked dependency chains. Any full proposal for this approach should therefore describe how this issue will be handled.

3.2 Mark Range of Code

Ranges of code could be marked by pragmas, through use of C preprocessor symbols, or via other ad-hoc means. However, again, software-engineering considerations would likely cause many such projects to mark all the translation units, which would of course result in the same dependency-chain-tracing complexity as would unmarked dependency chains. Any full proposal for this approach should therefore describe how this issue will be handled.

3.3 Mark Functions

Functions containing dependency chains could be marked with an attribute (for example, something like `[[function_carries_dependencies]]`) or a keyword (for example, something like `_FunctionCarriesDependencies`).

Proper use of this approach eliminates issues with dependencies passing through dependency-unaware code: Simply mark the relevant functions. However, although there are many software-engineering reasons for preferring small functions, the fact remains that large functions are not uncommon in production code. Large marked functions of course result in similar dependency-chain-tracing complexity as would unmarked code, so any full proposal for this approach should describe how this tracing will be handled.

3.4 Mark Objects

This class of proposals marks the objects that are to carry dependencies. These objects must be of pointer type. Note that implementations requiring point-to-point associations between each `memory_order_consume` load and its corresponding dependent memory references can generate these associations based on the operations carried out on a given marked object.

3.4.1 Attribute

This approach, suggested by Clark Nelson, generalizes the `[[carries_dependency]]` attribute specified in the C++11 standard so that it applies to objects, including variables, formal parameters, return values, and class members. This paper further modifies this proposed attribute so as to also restrict it to pointer-like objects.

There have been some objections to attributes on the grounds that attributes are not supposed to change program semantics, but no consensus as to whether or not this objection is substantive.

The changes to the examples from Section 2 are similar to those shown in Section 3.4.3.

3.4.2 Type Qualifier

This approach, put forward by Torvald Riegel in response to Linus Torvalds's spirited criticisms of the current C11 and C++11 wording, introduces a new `value_dep_preserving` type qualifier. Objects marked with this type qualifier carry dependencies.

Again, the changes to the examples from Section 2 are similar to those shown in Section 3.4.3.

3.4.3 Object Modifier

This approach uses a keyword that does not participate in type checking, for example, a `_Carries_dependency` keyword. This might be treated in a manner similar to a storage class. It need not necessarily interact with the type system.

Figures 11–19 show how object modifiers can be applied to each of the examples introduced in Section 2. These changes are straightforward markings of local variables, function parameters, and return-value types. Object modifiers therefore easily support the use cases in the Linux kernel.[1]

[1] Give or take a strong distaste for any sort of marking scheme on the part of numerous Linux-kernel community members.

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  assert(p->a != 43);
  rcu_assign_pointer(gp, p);
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  if (p)
    p->a = 43;
}
```

Figure 11: Object Modifier: Simple Case

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

void
thread1_help(rcutest _Carries_dependency *q)
{
  if (q)
    assert(q->a == 42);
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  thread1_help(p);
}
```

Figure 12: Object Modifier: In via Function Parameter

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

rcutest _Carries_dependency *thread1_help()
{
  return rcu_dereference(gp);
}

void thread1()
{
  rcutest _Carries_dependency *p = thread1_help();
  if (p)
    assert(p->a == 42);
}
```

Figure 13: Object Modifier: Out via Function Return

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  rcu_assign_pointer(gp, p);

  p = new rcutest();
  p->a = 43;
  rcu_assign_pointer(gsgp, p);
}

rcutest _Carries_dependency *
thread1_help(rcutest _Carries_dependency *p)
{
  if (p)
    assert(p->a == 42);
  return rcu_dereference(gsgp);
}

void thread1(void)
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  p = thread1_help(p);
  if (p)
    assert(p->a == 43);
}
```

Figure 14: Object Modifier: In and Out, But Different Chains

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

void
thread1_help1(rcutest _Carries_dependency *q)
{
  if (q)
    assert(q->a == 42);
}

void
thread1_help2(rcutest _Carries_dependency *q)
{
  if (q)
    assert(q->a != 43);
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  thread1_help1(p);
  thread1_help2(p);
}
```

Figure 15: Object Modifier: Chain Fanning Out

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  rcu_assign_pointer(gp, p);
  rcutest1 *p1 = new rcutest1();
  p1->a = 41;
  p1->rt.a = 42;
  rcu_assign_pointer(g1p, p1);
}

void
thread1_help(rcutest _Carries_dependency *q)
{
  if (q)
    assert(q->a == 42);
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  thread1_help(p);
}

void thread2()
{
  rcutest1 _Carries_dependency *p1 = rcu_dereference(g1p);
  thread1_help(&p1->rt);
}
```

Figure 16: Object Modifier: Chain Fanning In

3.4.4 Template

This approach, suggested off-list by JF Bastien, creates a `depending_ptr`[2] template to which a pointer-like type is passed. This approach allows implementers considerable freedom, as they can hook into the `->` and `*` if need be, and also use the C++ `delete` keyword to prohibit problematic operations. Implementations that might nevertheless carry out aggressive optimizations that might break dependencies even for the non-problematic operations might need to implement this template class in a manner similar to the atomics template classes.

This approach would need to be augmented with a non-template solution for C, for example, the object-modifier approach from Section 3.4.3. Implementations that support both C and C++ would presumably relate Section 3.4.3's keyword to the templates in this section in a manner similar to that used for atomics.

Figure 20 shows the resulting template declaration,

[2] Arbitrarily chosen name with no Google hits.

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  p->b = 43;
  rcu_assign_pointer(gp, p);
  rcutest1 *p1 = new rcutest1();
  p1->a = 41;
  p1->rt.a = 42;
  p1->rt.b = 43;
  rcu_assign_pointer(g1p, p1);
}

void
thread1a_help(rcutest _Carries_dependency *q)
{
  assert(q->a == 42);
}

void
thread1b_help(rcutest _Carries_dependency *q)
{
  assert(q->b == 43);
}

void
thread1_help(rcutest _Carries_dependency *q)
{
  if (q) {
    thread1a_help(q);
    thread1b_help(q);
  }
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  thread1_help(p);
}

void thread2()
{
  rcutest1 _Carries_dependency *p1 = rcu_dereference(g1p);
  thread1_help(&p1->rt);
}
```

Figure 17: Object Modifier: Chain Fanning In and Out

```
void thread0()
{
  struct rcutest *p = new rcutest();
  p->a = 42;
  p->b = 43;
  rcu_assign_pointer(gp, p);
  struct rcutest1 *p1 = new rcutest1();
  p1->a = 41;
  p1->rt.a = 42;
  p1->rt.b = 43;
  rcu_assign_pointer(g1p, p1);
}

#ifdef FOO
void
thread1a_help(rcutest _Carries_dependency *q)
{
  assert(q->a == 42);
}
#endif

void
thread1b_help(rcutest _Carries_dependency *q)
{
  assert(q->b == 43);
}

void
thread1_help(rcutest _Carries_dependency *q)
{
  if (q) {
#ifdef FOO
    thread1a_help(q);
#endif
    thread1b_help(q);
  }
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  thread1_help(p);
}

void thread2()
{
  rcutest1 _Carries_dependency *p1 = rcu_dereference(g1p);
  thread1_help(&p1->rt);
}
```

Figure 18: Object Modifier: Conditional Compilation of Chain Endpoints

```
void thread0()
{
  rcutest *p = new rcutest();
  p->a = 42;
  assert(p->a != 43);
  rcu_assign_pointer(gp, p);
}

void thread1()
{
  rcutest _Carries_dependency *p = rcu_dereference(gp);
  if (p) {
    assert(p->a == 42);
    spin_lock(&p->lock);
    p = std::kill_dependency(p);
    p->a++;
    spin_unlock(&p->lock);
  }
}
```

Figure 19: Object Modifier: Handoff to Locking

each member function of which has a straightforward definition. Note especially that the relational operators are defined in terms of the `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()` functions shown in Figure 21, so that as long as the first argument to a relational operator is of type `class depending_ptr<T>`, pointers may be safely compared without risk of breaking dependency chains.[3] In addition, the operators that cannot be safely applied to dependency-bearing pointers are omitted.[4] Finally, Figure 22 shows how the Linux-kernel-style `rcu_dereference()` and `rcu_assign_pointer()` macros could be implemented given this templated approach.

Figures 23–31 show how templates can be applied to each of the examples introduced in Section 2. As with the object-modifier approach in Section 3.4.3, these changes are straightforward markings of local variables, function parameters, and return-value types.

Full source code for a prototype implementation (and for this paper) may be downloaded from

```
template<typename T>
class depending_ptr {
public:
  typedef T* pointer;
  typedef T element_type;

  // Constructors
  constexpr depending_ptr() noexcept;
  explicit depending_ptr(T* v) noexcept;
  depending_ptr(nullptr_t) noexcept;
  depending_ptr(const depending_ptr &d) noexcept;
  depending_ptr(const depending_ptr &&d) noexcept;

  // Assignment
  depending_ptr& operator=(pointer p) noexcept;
  depending_ptr& operator=(const depending_ptr &d) noexcept;
  depending_ptr& operator=(const depending_ptr &&d) noexcept;
  depending_ptr& operator=(nullptr_t) noexcept;

  // Modifiers
  void swap(depending_ptr& d) noexcept;

  // Unary operators
  // No operator!
  // No prefix bitwise complement operator
  element_type operator*() noexcept;
  pointer operator->() noexcept;
  depending_ptr<element_type> operator++();
  depending_ptr<element_type> operator++(int);
  depending_ptr<element_type> operator--();
  depending_ptr<element_type> operator--(int);
  pointer get() const noexcept;
  explicit operator bool();
  element_type operator[](size_t);

  // Binary relational operators
  bool operator==(depending_ptr v) noexcept;
  bool operator!=(depending_ptr v) noexcept;
  bool operator>(depending_ptr v) noexcept;
  bool operator>=(depending_ptr v) noexcept;
  bool operator<(depending_ptr v) noexcept;
  bool operator<=(depending_ptr v) noexcept;
  bool operator==(pointer v) noexcept;
  bool operator!=(pointer v) noexcept;
  bool operator>(pointer v) noexcept;
  bool operator>=(pointer v) noexcept;
  bool operator<(pointer v) noexcept;
  bool operator<=(pointer v) noexcept;

  // Other binary operators
  depending_ptr<T> operator+(size_t idx);
  depending_ptr<T> operator+=(size_t idx);
  depending_ptr<T> operator-(size_t idx);
  depending_ptr<T> operator-=(size_t idx);

private:
  pointer dp_rep;
};
```

Figure 20: Template: Declaration

[3] That said, in the prototype implementation, these are not intrinsics, but rather separately compiled functions. In the absence of link-time optimizations, separate compilation preserves dependency chains in most implementations.

[4] The number of pointer-tagging algorithms should motivate allowing bitwise operations on dependency-bearing pointers, but this should be handled separately.

```
bool pointer_cmp_eq_dep(void *p, void *q) noexcept;
bool pointer_cmp_ne_dep(void *p, void *q) noexcept;
bool pointer_cmp_gt_dep(void *p, void *q) noexcept;
bool pointer_cmp_ge_dep(void *p, void *q) noexcept;
bool pointer_cmp_lt_dep(void *p, void *q) noexcept;
bool pointer_cmp_le_dep(void *p, void *q) noexcept;
```

Figure 21: Dependency-Preserving Comparisons

```
template<typename T>
depending_ptr<T> rcu_consume(std::atomic<T*> *p)
{
  volatile std::atomic<typename
           depending_ptr<T>::pointer> *q = p;
  // Change to memory_order_consume once it is fixed
  depending_ptr<T> temp(q->load(std::memory_order_relaxed));

  return temp;
}

template<typename T>
depending_ptr<T> rcu_consume(T *p)
{
  // Alternatively, could cast p to volatile atomic...
  depending_ptr<T> temp(*(T *volatile *)&p);

  return temp;
}

template<typename T>
T* rcu_store_release(std::atomic<T*> *p, T *v)
{
  p->store(v, std::memory_order_release);
  return v;
}

template<typename T>
T* rcu_store_release(T **p, T *v)
{
  // Alternatively, could cast p to volatile atomic...
  atomic_thread_fence(std::memory_order_release);
  *((volatile T **)p) = v;
  return v;
}

// Linux-kernel compatibility macros, not for atomics
#define rcu_dereference(p) rcu_consume(p)
#define rcu_assign_pointer(p, v) rcu_store_release(&(p), v)
```

Figure 22: Dependency-Preserving Release and Consume

```
void *thread0(void *unused)
{
  rcutest *p;

  p = new rcutest();
  assert(p);
  p->a = 42;
  assert(p->a != 43);
  rcu_store_release(&gp, p);
  return nullptr;
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  if (p)
    p->a = 43;
  return nullptr;
}
```

Figure 23: Template: Simple Case

```
void *thread0(void *unused)
{
  rcutest *p;

  p = new rcutest();
  assert(p);
  p->a = 42;
  rcu_store_release(&gp, p);
  return nullptr;
}

void
thread1_help(depending_ptr<rcutest> q)
{
  if (q)
    assert(q->a == 42);
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  thread1_help(p);
  return nullptr;
}
```

Figure 24: Template: In via Function Parameter

```
void *thread0(void *unused)
{
  rcutest *p;

  p = new rcutest();
  assert(p);
  p->a = 42;
  rcu_store_release(&gp, p);
  return nullptr;
}

depending_ptr<rcutest> thread1_help()
{
  return rcu_consume(&gp);
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = thread1_help();
  if (p)
    p->a = 43;
  return nullptr;
}
```

Figure 25: Template: Out via Function Return

`https://github.com/paulmckrcu/2016markconsume.git`.

3.5 Mark Root/Leaf Pairs

These approaches create point-to-point associations between `memory_order_consume` loads and the memory references that depend on them. Function calls can be handled by using the arguments of the function call and the function parameters as intermediate points in the association. Function returns can be handled by using the function return declaration and the function return value.

However, these point-to-point associations are required to gracefully handle bushy dependency trees, dependency trees that fan both in and out, and conditional compilation. Any scheme that relies on directly referencing a specific location in the source code will fall afoul of these requirements.

One approach is to use a unique identifier for each dependency tree, and associate each relevant point in the code with the corresponding identifiers.

Note that the root-leaf information could in theory be extracted by the compiler based on object markings (see Section 3.4).

```
void *thread0(void *unused)
{
  rcutest *p;

  p = new rcutest();
  assert(p);
  p->a = 42;
  rcu_store_release(&gp, p);

  p = new rcutest();
  assert(p);
  p->a = 43;
  rcu_store_release(&gsgp, p);

  return nullptr;
}

depending_ptr<rcutest>
thread1_help(depending_ptr<rcutest> p)
{
  if (p)
    assert(p->a == 42);
  return rcu_consume(&gsgp);
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  p = thread1_help(p);
  if (p)
    assert(p->a == 43);
  return nullptr;
}
```

Figure 26: Template: In and Out, But Different Chains

```
void *thread0(void *unused)
{
  rcutest *p;

  p = new rcutest();
  p->a = 42;
  rcu_store_release(&gp, p);
  return nullptr;
}

void thread1_help1(depending_ptr<rcutest> q)
{
  if (q)
    assert(q->a == 42);
}

void thread1_help2(depending_ptr<rcutest> q)
{
  if (q)
    assert(q->a != 43);
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  thread1_help1(p);
  thread1_help2(p);
  return nullptr;
}
```

Figure 27: Template: Chain Fanning Out

```
void *thread0(void *unused)
{
  rcutest *p;
  rcutest1 *p1;

  p = new rcutest();
  p->a = 42;
  rcu_store_release(&gp, p);

  p1 = new rcutest1();
  p1->a = 41;
  p1->rt.a = 42;
  rcu_store_release(&g1p, p1);

  return nullptr;
}

void thread1_help(depending_ptr<rcutest> q)
{
  if (q)
    assert(q->a == 42);
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  thread1_help(p);
  return nullptr;
}

void *thread2(void *unused)
{
  depending_ptr<rcutest1> p1;

  p1 = rcu_consume(&g1p);
  thread1_help(depending_ptr<rcutest>(&p1->rt));
  return nullptr;
}
```

Figure 28: Template: Chain Fanning In

```
void *thread0(void *unused)
{
  rcutest *p;
  rcutest1 *p1;

  p = new rcutest();
  assert(p);
  p->a = 42;
  p->b = 43;
  rcu_store_release(&gp, p);

  p1 = new rcutest1();
  assert(p1);
  p1->a = 41;
  p1->rt.a = 42;
  p1->rt.b = 43;
  rcu_store_release(&g1p, p1);

  return nullptr;
}

void thread1a_help(depending_ptr<rcutest> q)
{
  assert(q->a == 42);
}

void thread1b_help(depending_ptr<rcutest> q)
{
  assert(q->b == 43);
}

void thread1_help(depending_ptr<rcutest> q)
{
  if (q) {
    thread1a_help(q);
    thread1b_help(q);
  }
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  thread1_help(p);
  return nullptr;
}

void *thread2(void *unused)
{
  depending_ptr<rcutest1> p1;

  p1 = rcu_consume(&g1p);
  thread1_help(depending_ptr<rcutest>(&p1->rt));
  return nullptr;
}
```

Figure 29: Template: Chain Fanning In and Out

```
void *thread0(void *unused)
{
  rcutest *p;
  rcutest1 *p1;

  p = new rcutest();
  assert(p);
  p->a = 42;
  p->b = 43;
  rcu_store_release(&gp, p);

  p1 = new rcutest1();
  assert(p1);
  p1->a = 41;
  p1->rt.a = 42;
  p1->rt.b = 43;
  rcu_store_release(&g1p, p1);

  return nullptr;
}

#ifdef FOO
void thread1a_help(depending_ptr<rcutest> q)
{
  assert(q->a == 42);
}
#endif

void thread1b_help(depending_ptr<rcutest> q)
{
  assert(q->b == 43);
}

void thread1_help(depending_ptr<rcutest> q)
{
  if (q) {
#ifdef FOO
    thread1a_help(q);
#endif
    thread1b_help(q);
  }
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  thread1_help(p);
  return nullptr;
}

void *thread2(void *unused)
{
  depending_ptr<rcutest1> p1;

  p1 = rcu_consume(&g1p);
  thread1_help(depending_ptr<rcutest>(&p1->rt));
  return nullptr;
}
```

Figure 30: Template: Conditional Compilation of Chain Endpoints

```
void *thread0(void *unused)
{
  rcutest *p;

  p = new rcutest();
  p->a = 42;
  assert(p->a != 43);
  rcu_store_release(&gp, p);
  return nullptr;
}

void *thread1(void *unused)
{
  depending_ptr<rcutest> p;

  p = rcu_consume(&gp);
  if (p) {
    assert(p->a == 42);
    spin_lock(&p->lock);
    p = std::kill_dependency(p);
    p->a++;
    spin_unlock(&p->lock);
  }
  return nullptr;
}
```

Figure 31: Template: Handoff to Locking

4 Evaluation

Table 1 provides a rough comparison between the various marking methods, and also includes the unmarked option for comparison purposes.

For ease of compilation, the cells corresponding to methods that explicitly mark dependency chains or that don't require marking at all are left blank. Those that require tracing dependency chains throughout the full translation unit are marked "T" and those that limit the code in which tracing is required are marked "t".

For ease of modification, the cells corresponding to methods that either require no marking or that mark large-scale entities are left blank. Those that require marking the definitions of objects that carry dependencies are marked "o", and those require marking of individual accesses are marked "A".

Cells corresonding to those methods that precisely mark dependency chains are left blank, otherwise, they are marked "N".

For cross-function dependency chains, those methods that either support cross-function marking or that do not require such marking are left blank. Those that require manual consistency checks are

Mark:	Ease of Compilation	Ease of Modification	Precise Dependency Chains	Cross-Function Dependency Chains	Cross-Compilation-Unit Dependency Chains	C Compatibility	Formal Verification
Translation Unit	T		N		m		N
Range of Code	t		N	m	m		N
Functions	t		N	m	m		N
Objects							
Attribute		o		t	t	a	
Type Qualifier		o					N
Modifier		o		t	t		
Template		o				N	
Template+Modifier		o					
Root/Leaf		A		?	?	?	?
Nothing			N				N

Table 1: Dependency-Chain Marking Evaluation

marked "m", those that are amenable to consistency-check tooling are marked "t", and those that are not fleshed out sufficiently to tell are marked "?". These same markings are used for cross-compilation-unit dependency chains.

Cells corresponding to those methods supporting C compatibility are left blank. Those that would support C compatibility if C were to provide attributes are marked "a". Those that do not support C compatibility (at least not unless combined with some other method) are marked "N", and those that are not fleshed out sufficiently to tell are marked "?".

Cells corresponding to methods believed to support formal verification are left blank, those that are believed not to support formal verification are marked "N", and those that are not fleshed out sufficiently to tell are marked "?". Note that the object type qualifier could in theory support formal verification, but the specific proposal rules this out by requiring that the compiler treat `memory_order_consume` loads as potentially returning any value from the type.

Following the lead of C11 and C++11 atomics, the "Template+Modifier" row covers the combination of marking objects with template classes (for C++) and with an typed object modifier (for C), a combination that appears to be quite attractive. This should be further combined with a totally unmarked option for use by standalone projects such as the Linux kernel.

The best possible method would have a row of all blank cells.

5 Summary

This paper reviewed the 2016 discussions of `memory_order_consume` that took place at the Jacksonville meeting, presented several representative use cases, listed evaluation criteria, presented a number of marking proposals, and provided a comparative evaluation. The paper presents two of the marking proposals in depth, including code for the representative use cases.

We recommend a combination of typed object modifier (for C compatibility) and a template class (for C++), which is similar to the approach used by atomics. For standalone applications such as the Linux kernel, there should additionally be an unmarked option, where the implementation assumes that everything that could legally marked is so marked.

References

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