PROPER CARE AND FEEDING OF RETURN VALUES FROM rcu_dereference()

Most of the time, you can use values from rcu_dereference() or one of the similar primitives without worries. Dereferencing (prefix "*"), field selection ("->"), assignment ("="), address-of ("&"), addition and subtraction of constants, and casts all work quite naturally and safely.

It is nevertheless possible to get into trouble with other operations. Follow these rules to keep your RCU code working properly:

- You must use one of the rcu_dereference() family of primitives to load an RCU-protected pointer, otherwise CONFIG_PROVE_RCU will complain. Worse yet, your code can see random memory-corruption bugs due to games that compilers and DEC Alpha can play. Without one of the rcu_dereference() primitives, compilers can reload the value, and won't your code have fun with two different values for a single pointer! Without rcu_dereference(), DEC Alpha can load a pointer, dereference that pointer, and return data preceding initialization that preceded the store of the pointer.

  In addition, the volatile cast in rcu_dereference() prevents the compiler from deducing the resulting pointer value. Please see the section entitled "EXAMPLE WHERE THE COMPILER KNOWS TOO MUCH" for an example where the compiler can in fact deduce the exact value of the pointer, and thus cause misordering.

- In the special case where data is added but is never removed while readers are accessing the structure, READ_ONCE() may be used instead of rcu_dereference(). In this case, use of READ_ONCE() takes on the role of the lockless_dereference() primitive that was removed in v4.15.

- You are only permitted to use rcu_dereference on pointer values. The compiler simply knows too much about integral values to trust it to carry dependencies through integer operations. There are a very few exceptions, namely that you can temporarily cast the pointer to uintptr_t in order to:

  - Set bits and clear bits down in the must-be-zero low-order bits of that pointer. This clearly means that the pointer must have alignment constraints, for example, this does *not* work in general for char* pointers.
  - XOR bits to translate pointers, as is done in some classic buddy-allocator algorithms.

  It is important to cast the value back to pointer before doing much of anything else with it.

- Avoid cancellation when using the "+" and "-" infix arithmetic operators. For example, for a given variable "x", avoid "(x-(uintptr_t)x)" for char* pointers. The compiler is within its rights to substitute zero for this sort of expression, so that subsequent accesses no longer depend on the rcu_dereference(), again possibly resulting in bugs due to misordering.

  Of course, if "p" is a pointer from rcu_dereference(), and "a" and "b" are integers that happen to be equal, the expression "p+a-b" is safe because its value still necessarily depends on the rcu_dereference(), thus maintaining proper ordering.

- If you are using RCU to protect JITed functions, so that the "()" function-invocation operator is applied to a value obtained (directly or indirectly) from rcu_dereference(), you may need to interact directly with the hardware to flush instruction caches. This issue arises on some systems when a newly JITed function is using the same memory that was used by an earlier JITed function.

- Do not use the results from relational operators ("==", "!=", ">", ">=", "<", or "<=") when dereferencing. For example, the following (quite strange) code is buggy:

  ```
  int *p;
  int *q;

  ...

  p = rcu_dereference(gp)
  q = &global_q;
  q += p > &oom_p;
  r1 = *q;  /* BUGGY!!! */
  ```

  As before, the reason this is buggy is that relational operators are often compiled using branches. And as before, although weak-memory machines such as ARM or PowerPC do order stores after such branches, but can speculate loads, which can again result in misordering bugs.

- Be very careful about comparing pointers obtained from rcu_dereference() against non-NULL values. As Linus Torvalds explained, if the two pointers are equal, the compiler could substitute the pointer you are comparing against for the pointer obtained from rcu_dereference(). For example:

  ```
  p = rcu_dereference(gp);
  if (p == &default_struct)
          do_default(p->a);
  ```

  Because the compiler now knows that the value of "p" is exactly the address of the variable "default_struct", it is free to transform this code into the following:

```
p = rcu_dereference(gp);
if (p == &default_struct)
        do_default(default_struct.a);
```

On ARM and Power hardware, the load from "default_struct.a" can now be speculated, such that it might happen before the rcu_dereference(). This could result in bugs due to misordering.

However, comparisons are OK in the following cases:

- The comparison was against the NULL pointer. If the compiler knows that the pointer is NULL, you had better not be dereferencing it anyway. If the comparison is non-equal, the compiler is none the wiser. Therefore, it is safe to compare pointers from rcu_dereference() against NULL pointers.
- The pointer is never dereferenced after being compared. Since there are no subsequent dereferences, the compiler cannot use anything it learned from the comparison to reorder the non-existent subsequent dereferences. This sort of comparison occurs frequently when scanning RCU-protected circular linked lists.

  Note that if checks for being within an RCU read-side critical section are not required and the pointer is never dereferenced, rcu_access_pointer() should be used in place of rcu_dereference().
- The comparison is against a pointer that references memory that was initialized "a long time ago." The reason this is safe is that even if misordering occurs, the misordering will not affect the accesses that follow the comparison. So exactly how long ago is "a long time ago"? Here are some possibilities:
  - Compile time.
  - Boot time.
  - Module-init time for module code.
  - Prior to kthread creation for kthread code.
  - During some prior acquisition of the lock that we now hold.
  - Before mod_timer() time for a timer handler.

  There are many other possibilities involving the Linux kernel's wide array of primitives that cause code to be invoked at a later time.
- The pointer being compared against also came from rcu_dereference(). In this case, both pointers depend on one rcu_dereference() or another, so you get proper ordering either way.

  That said, this situation can make certain RCU usage bugs more likely to happen. Which can be a good thing, at least if they happen during testing. An example of such an RCU usage bug is shown in the section titled "EXAMPLE OF AMPLIFIED RCU-USAGE BUG".
- All of the accesses following the comparison are stores, so that a control dependency preserves the needed ordering. That said, it is easy to get control dependencies wrong. Please see the "CONTROL DEPENDENCIES" section of Documentation/memory-barriers.txt for more details.
- The pointers are not equal *and* the compiler does not have enough information to deduce the value of the pointer. Note that the volatile cast in rcu_dereference() will normally prevent the compiler from knowing too much.

  However, please note that if the compiler knows that the pointer takes on only one of two values, a not-equal comparison will provide exactly the information that the compiler needs to deduce the value of the pointer.

- Disable any value-speculation optimizations that your compiler might provide, especially if you are making use of feedback-based optimizations that take data collected from prior runs. Such value-speculation optimizations reorder operations by design.

  There is one exception to this rule: Value-speculation optimizations that leverage the branch-prediction hardware are safe on strongly ordered systems (such as x86), but not on weakly ordered systems (such as ARM or Power). Choose your compiler command-line options wisely!

EXAMPLE OF AMPLIFIED RCU-USAGE BUG

Because updaters can run concurrently with RCU readers, RCU readers can see stale and/or inconsistent values. If RCU readers need fresh or consistent values, which they sometimes do, they need to take proper precautions. To see this, consider the following code fragment:

```
struct foo {
        int a;
        int b;
        int c;
};
struct foo *gp1;
struct foo *gp2;

void updater(void)
{
        struct foo *p;
```

```
        p = kmalloc(...);
        if (p == NULL)
                deal_with_it();
        p->a = 42;  /* Each field in its own cache line. */
        p->b = 43;
        p->c = 44;
        rcu_assign_pointer(gp1, p);
        p->b = 143;
        p->c = 144;
        rcu_assign_pointer(gp2, p);
}

void reader(void)
{
        struct foo *p;
        struct foo *q;
        int r1, r2;

        p = rcu_dereference(gp2);
        if (p == NULL)
                return;
        r1 = p->b;  /* Guaranteed to get 143. */
        q = rcu_dereference(gp1);  /* Guaranteed non-NULL. */
        if (p == q) {
                /* The compiler decides that q->c is same as p->c. */
                r2 = p->c; /* Could get 44 on weakly order system. */
        }
        do_something_with(r1, r2);
}
```

You might be surprised that the outcome (r1 == 143 && r2 == 44) is possible, but you should not be. After all, the updater might have been invoked a second time between the time reader() loaded into "r1" and the time that it loaded into "r2". The fact that this same result can occur due to some reordering from the compiler and CPUs is beside the point.

But suppose that the reader needs a consistent view?

Then one approach is to use locking, for example, as follows:

```
struct foo {
        int a;
        int b;
        int c;
        spinlock_t lock;
};
struct foo *gp1;
struct foo *gp2;

void updater(void)
{
        struct foo *p;

        p = kmalloc(...);
        if (p == NULL)
                deal_with_it();
        spin_lock(&p->lock);
        p->a = 42;  /* Each field in its own cache line. */
        p->b = 43;
        p->c = 44;
        spin_unlock(&p->lock);
        rcu_assign_pointer(gp1, p);
        spin_lock(&p->lock);
        p->b = 143;
        p->c = 144;
        spin_unlock(&p->lock);
        rcu_assign_pointer(gp2, p);
}

void reader(void)
{
        struct foo *p;
        struct foo *q;
        int r1, r2;

        p = rcu_dereference(gp2);
        if (p == NULL)
                return;
        spin_lock(&p->lock);
        r1 = p->b;  /* Guaranteed to get 143. */
        q = rcu_dereference(gp1);  /* Guaranteed non-NULL. */
        if (p == q) {
                /* The compiler decides that q->c is same as p->c. */
```

```
                r2 = p->c; /* Locking guarantees r2 == 144. */
        }
        spin_unlock(&p->lock);
        do_something_with(r1, r2);
}
```

As always, use the right tool for the job!

EXAMPLE WHERE THE COMPILER KNOWS TOO MUCH

If a pointer obtained from rcu_dereference() compares not-equal to some other pointer, the compiler normally has no clue what the value of the first pointer might be. This lack of knowledge prevents the compiler from carrying out optimizations that otherwise might destroy the ordering guarantees that RCU depends on. And the volatile cast in rcu_dereference() should prevent the compiler from guessing the value.

But without rcu_dereference(), the compiler knows more than you might expect. Consider the following code fragment:

```
struct foo {
        int a;
        int b;
};
static struct foo variable1;
static struct foo variable2;
static struct foo *gp = &variable1;

void updater(void)
{
        initialize_foo(&variable2);
        rcu_assign_pointer(gp, &variable2);
        /*
         * The above is the only store to gp in this translation unit,
         * and the address of gp is not exported in any way.
         */
}

int reader(void)
{
        struct foo *p;

        p = gp;
        barrier();
        if (p == &variable1)
                return p->a; /* Must be variable1.a. */
        else
                return p->b; /* Must be variable2.b. */
}
```

Because the compiler can see all stores to "gp", it knows that the only possible values of "gp" are "variable1" on the one hand and "variable2" on the other. The comparison in reader() therefore tells the compiler the exact value of "p" even in the not-equals case. This allows the compiler to make the return values independent of the load from "gp", in turn destroying the ordering between this load and the loads of the return values. This can result in "p->b" returning pre-initialization garbage values.

In short, rcu_dereference() is *not* optional when you are going to dereference the resulting pointer.

WHICH MEMBER OF THE rcu_dereference() FAMILY SHOULD YOU USE?

First, please avoid using rcu_dereference_raw() and also please avoid using rcu_dereference_check() and rcu_dereference_protected() with a second argument with a constant value of 1 (or true, for that matter). With that caution out of the way, here is some guidance for which member of the rcu_dereference() to use in various situations:

1. If the access needs to be within an RCU read-side critical section, use rcu_dereference(). With the new consolidated RCU flavors, an RCU read-side critical section is entered using rcu_read_lock(), anything that disables bottom halves, anything that disables interrupts, or anything that disables preemption.
2. If the access might be within an RCU read-side critical section on the one hand, or protected by (say) my_lock on the other, use rcu_dereference_check(), for example:

   ```
   p1 = rcu_dereference_check(p->rcu_protected_pointer,
                              lockdep_is_held(&my_lock));
   ```

3. If the access might be within an RCU read-side critical section on the one hand, or protected by either my_lock or your_lock on the other, again use rcu_dereference_check(), for example:

   ```
   p1 = rcu_dereference_check(p->rcu_protected_pointer,
                              lockdep_is_held(&my_lock) ||
   ```

```
                                lockdep_is_held(&your_lock));
```

4. If the access is on the update side, so that it is always protected by my_lock, use rcu_dereference_protected():

   ```
   p1 = rcu_dereference_protected(p->rcu_protected_pointer,
                                  lockdep_is_held(&my_lock));
   ```

   This can be extended to handle multiple locks as in #3 above, and both can be extended to check other conditions as well.

5. If the protection is supplied by the caller, and is thus unknown to this code, that is the rare case when rcu_dereference_raw() is appropriate. In addition, rcu_dereference_raw() might be appropriate when the lockdep expression would be excessively complex, except that a better approach in that case might be to take a long hard look at your synchronization design. Still, there are data-locking cases where any one of a very large number of locks or reference counters suffices to protect the pointer, so rcu_dereference_raw() does have its place.

   However, its place is probably quite a bit smaller than one might expect given the number of uses in the current kernel. Ditto for its synonym, rcu_dereference_check(... , 1), and its close relative, rcu_dereference_protected(... , 1).

SPARSE CHECKING OF RCU-PROTECTED POINTERS

The sparse static-analysis tool checks for direct access to RCU-protected pointers, which can result in "interesting" bugs due to compiler optimizations involving invented loads and perhaps also load tearing. For example, suppose someone mistakenly does something like this:

```
p = q->rcu_protected_pointer;
do_something_with(p->a);
do_something_else_with(p->b);
```

If register pressure is high, the compiler might optimize "p" out of existence, transforming the code to something like this:

```
do_something_with(q->rcu_protected_pointer->a);
do_something_else_with(q->rcu_protected_pointer->b);
```

This could fatally disappoint your code if q->rcu_protected_pointer changed in the meantime. Nor is this a theoretical problem: Exactly this sort of bug cost Paul E. McKenney (and several of his innocent colleagues) a three-day weekend back in the early 1990s.

Load tearing could of course result in dereferencing a mashup of a pair of pointers, which also might fatally disappoint your code.

These problems could have been avoided simply by making the code instead read as follows:

```
p = rcu_dereference(q->rcu_protected_pointer);
do_something_with(p->a);
do_something_else_with(p->b);
```

Unfortunately, these sorts of bugs can be extremely hard to spot during review. This is where the sparse tool comes into play, along with the "__rcu" marker. If you mark a pointer declaration, whether in a structure or as a formal parameter, with "__rcu", which tells sparse to complain if this pointer is accessed directly. It will also cause sparse to complain if a pointer not marked with "__rcu" is accessed using rcu_dereference() and friends. For example, ->rcu_protected_pointer might be declared as follows:

```
struct foo __rcu *rcu_protected_pointer;
```

Use of "__rcu" is opt-in. If you choose not to use it, then you should ignore the sparse warnings.