N4036: Towards Implementation and Use of `memory_order_consume`

Doc. No.: WG21/N4036
Date: 2014-05-22
Reply to: Paul E. McKenney, Torvald Riegel, and Jeff Preshing
Email: paulmck@linux.vnet.ibm.com, triegel@redhat.com, jeff@preshing.com

Other contributors: Alec Teal, David Howells, David Lang, George Spelvin, Jeff Law, Joseph S. Myers, Lawrence Crowl, Linus Torvalds, Mark Batty, Michael Matz, Olivier Giroux, Peter Sewell, Peter Zijlstra, Ramana Radhakrishnan, Richard Biener, Will Deacon, ...

May 26, 2014

1 Introduction

The most obscure member of the C11 and C++11 `memory_order enum` seems to be `memory_order_consume` [27]. The purpose of `memory_order_consume` is to allow reading threads to correctly traverse linked data structures without the need for locks, atomic instructions, or (with the exception of DEC Alpha) memory-fence instructions, even though new elements are being inserted into these linked structures before, during, and after the traversal. Without `memory_order_consume`, both the compiler and (again, in the case of DEC Alpha) the CPU would be within their rights to carry out aggressive data-speculation optimizations that would permit readers to see pre-initialization values in the newly added data elements. The purpose of `memory_order_consume` is to prevent these optimizations.

Of course, `memory_order_acquire` may be used as a substitute for `memory_order_consume`, however doing so results in costly explicit memory-fence instructions (or, where available, load-acquire instructions) on weakly ordered systems such as ARM, Itanium, and PowerPC [9, 3, 12, 13]. These systems enforce dependency ordering in hardware, in other words, if the address used by one memory-reference instruction depends on the value from a preceding load instruction, the hardware forces that earlier load to complete before the later memory-reference instruction commences.[1] Similarly, if the data to be stored by a given store instruction depends on the value from a preceding load instruction, the hardware again forces that earlier load to complete before the later store instruction commences. Recent software tools for ARM and PowerPC can help explicate their memory models [19, 1, 2]. Note that strongly ordered systems like x86, IBM mainframe, and SPARC TSO enforce dependency ordering as a side effect of the fact that they do not reorder loads with subsequent memory references. Therefore, `memory_order_consume` is beneficial on hot code paths, removing the need for hardware ordering instructions for weakly ordered systems and permitting additional compiler optimizations on strongly ordered systems.

When implementing concurrent insertion-only data structures, a few of which are found in the Linux

[1] But please note that hardware can and does take advantage of the as-if rule, just as compilers do.

kernel, `memory_order_consume` is all that is required. However, most data structures also require removal of data elements. Such removal requires that the thread removing the data element wait for all readers to release their references to it before reclaiming that element. The traditional way to do this is via garbage collectors (GCs), which have been available for more than half a century [15] and which are now available even for C and C++ [4]. Another way to wait for readers is to use read-copy update (RCU) [23, 21], which explicitly marks read-side regions of code and provides primitives that wait for all pre-existing readers to complete. RCU is gaining significant use both within the Linux kernel [16] and outside of it [6, 5, 8, 14, 28].

Despite the growing number of `memory_order_consume` use cases, there are no known high-performance implementations of `memory_order_consume` loads in any C11 or C++11 environments. This situation suggests that some change is in order: After all, if the standard does not support this use case, the corresponding users can be expected to continue to exploit whatever implementation-specific facilities provide the required functionality. This document therefore provides a brief overview of RCU in Section 2 and surveys `memory_order_consume` use cases within the Linux kernel in Section 3. Section 4 looks at how dependency ordering is currently supported in pre-C11 implementations, and then Section 5 looks at possible ways to support those use cases in existing C11 and C++11 implementations, followed by some thoughts on incremental paths towards official support of these use cases in the standards. Section 6 lists some weaknesses in the current C11 and C++11 specification of dependency ordering, and finally Section 7 outlines a few possible alternative dependency-ordering specifications.

Note: SC22/WG14 liason issue.

2 Introduction to RCU

The RCU synchronization mechanism is often used as a replacement for reader-writer locking because RCU avoids the high-overhead cache thrashing that is characteristic of many common reader-writer-locking implementations. RCU is based on three fundamental concepts:

1. Light-weight in-memory publish-subscribe operation.
2. Operation that waits for pre-existing readers.
3. Maintaining multiple versions of data to avoid disrupting old readers that are still referencing old versions.

We would like to use C11's and C++11's `memory_order_consume` to implement RCU's lightweight subscribe operation, `rcu_dereference()`. We assume that `rcu_dereference()` is a good example of how developers would exploit the dependency-ordering feature of weakly ordered systems, so we look to `rcu_dereference()` as an indication of the semantics that `memory_order_consume` should have.

In one typical RCU use case, updaters publish new versions of a data structure while readers concurrently subscribe to whatever version is current at the time a given reader starts. Once all pre-existing readers complete, old versions can be reclaimed. This sort of use case may be a bit unfamiliar to many, but it is extremely effective in many situations, offering excellent performance, scalability, real-time latency, deadlock avoidance, and read-side composability. More details on RCU are readily available [8, 17, 18, 20, 21, 22, 24].

Figure 1 shows the growth of RCU usage over time within the Linux kernel, which is strong evidence of RCU's effectiveness. However, RCU is a specialized mechanism, so its use is much smaller than general-purpose techniques such as locking, as can be seen in Figure 2. It is unlikely that RCU's usage will ever approach that of locking because RCU coordinates only between readers and updaters, which means that some other mechanism is required to coordinate among concurrent updates. In the Linux kernel, that update-side mechanism is normally locking, although pretty much any synchronization mechanism may be used, including transactional memory [10, 11, 26].

However RCU is now being used in many situations where reader-writer locking would be used. Figure 3 shows that the use of reader-writer locking has

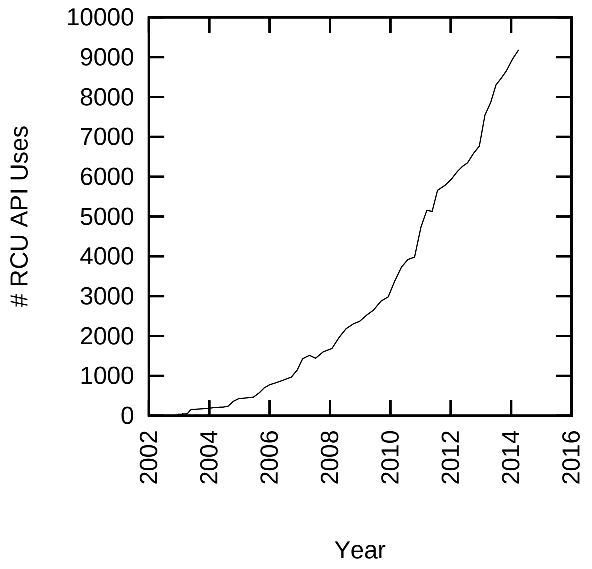

Figure 1: Growth of RCU Usage

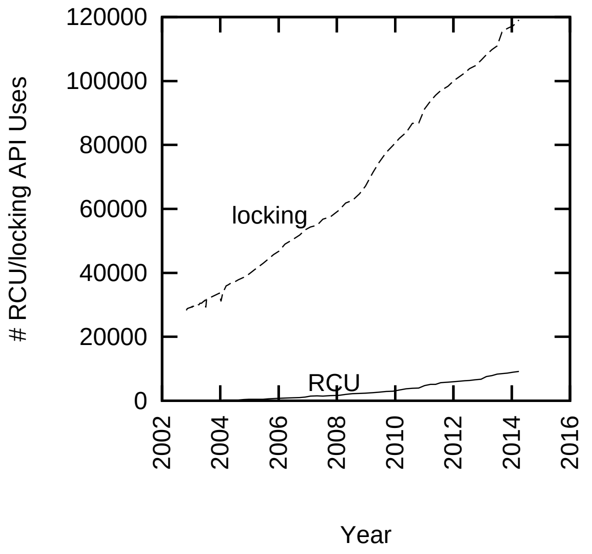

Figure 2: Growth of RCU Usage vs. Locking

changed little since RCU was introduced. This data suggests that RCU is at least as important to parallel software as is reader-writer locking.

In more recent years, a user-level library implementation of RCU has been available [7]. This library is now available for many platforms and has been included in a number of Linux distributions. It has been pressed into service for a number of open-source software projects, proprietary products, and reserch efforts.

Fully and fully performant C11/C++11 support for `memory_order_consume` is therefore quite important. However, good progress can often be made in the short term by focusing on the cases that are commonly used in practice rather than on the general case. The next section therefore takes a rough census of the Linux kernel's use of the `rcu_dereference()` family of primitives, which `memory_order_consume` is intended to implement.

3 Linux-Kernel Use Cases

Section 3.1 lists types of dependency chains in the Linux kernel, Section 3.2 lists operators used within these dependency chains, Section'3.3 lists operators that are considered to terminate dependency chains, and finally Section 3.4 surveys a longer-than-average (but by no means maximal) dependency chain that appears in the Linux kernel.

It is worth reviewing the relationship between `memory_order_acquire` and `memory_order_consume` loads, both of which interact with `memory_release_stores`.

A `memory_order_acquire` load is said to *synchronize with* a `memory_order_release` store if that load returns the value stored or in some special cases, some later value [27, 1.10p6-1.10p8]. When a `memory_order_acquire` load synchronizes with a `memory_order_release` store, any memory reference preceding the `memory_order_acquire` load will *happen before* any memory reference following the `memory_order_release` store [27, 1.10p11-1.10p12]. This property allows a linked structure to be locklessly traversed by using `memory_order_release` stores when updating pointers to reference new data elements and

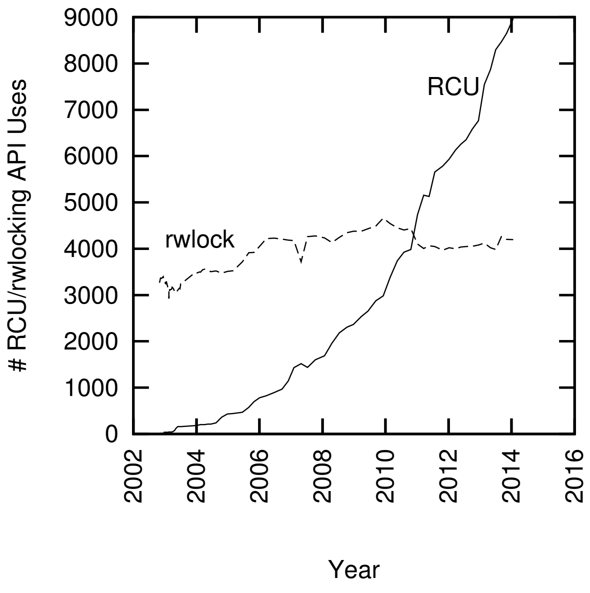

Figure 3: Growth of RCU Usage vs. Reader-Writer Locking

by using `memory_order_acquire` loads when loading pointers while locklessly traversing the data structure, as shown in Figure 4.

Unfortunately, a `memory_order_acquire` load requires expensive special load instructions or memory-fence instructions on weakly ordered systems such as ARM, Itanium, and PowerPC. Furthermore, in `traverse()`, the address of each `memory_order_acquire` load within the while loop depends on the value of the previous `memory_order_acquire` load.[2] Therefore, in this case, weakly ordered systems don't really need the special load instructions or the memory-fence instructions, as these systems can instead rely on the hardware-enforced dependency ordering.

This is the use case for `memory_order_consume`, which can be substituted for `memory_order_acquire` in cases where hardware dependency ordering applies. One such case is the preceding example, and Fig-

[2] The initial load on line 16 might well depend on an earlier load, but for simplicity, this example assumes that the initial `foo_head` structure is statically allocated, and thus not subject to updates.

```
 1 void new_element(struct foo **pp, int a)
 2 {
 3   struct foo *p = malloc(sizeof(*p));
 4
 5   if (!p)
 6     abort();
 7   p->a = a;
 8   atomic_store_explicit(pp, p, memory_order_release);
 9 }
10
11 int traverse(struct foo_head *ph)
12 {
13   int a = -1;
14   struct foo *p;
15
16   p = atomic_load_explicit(&ph->h, memory_order_acquire);
17   while (p != NULL) {
18     a = p->a;
19     p = atomic_load_explicit(&p->n, memory_order_acquire);
20   }
21   return a;
22 }
23


24
```

Figure 4: Release/Acquire Linked Structure Traversal

```
 1 void new_element(struct foo **pp, int a)
 2 {
 3   struct foo *p = malloc(sizeof(*p));
 4
 5   if (!p)
 6     abort();
 7   p->a = a;
 8   atomic_store_explicit(pp, p, memory_order_release);
 9 }
10
11 int traverse(struct foo_head *ph)
12 {
13   int a = -1;
14   struct foo *p;
15
16   p = atomic_load_explicit(&ph->h, memory_order_consume);
17   while (p != NULL) {
18     a = p->a;
19     p = atomic_load_explicit(&p->n, memory_order_consume);
20   }
21   return a;
22 }
23


24
```

Figure 5: Release/Consume Linked Structure Traversal

ure 5 shows that same example recast in terms of memory_order_consume. A memory_order_release store is *dependency ordered before* a memory_order_consume load when that load returns the value stored, or in some special cases, some later value [27, 1.10p1]. Then, if the load *carries a dependency* to some later memory reference [27, 1.10p9], any memory reference preceding the memory_order_release store will happen before that later memory reference [27, 1.10p9-1.10p12]. This means that when there is dependency ordering, memory_order_consume gives the same guarantees that memory_order_acquire does, but at lower cost.

On the other hand, memory_order_consume requires the compiler to track the carries-a-dependency relationships, with the set of such relationships headed by a given memory_order_consume load being called that load's *dependency chains.* It is quite possible that the complexity of implementing this capability has thus far prevented high-quality memory_order_consume implementations from appearing. It is therefore worthwhile to review use of dependency chains in practice in order to determine what types of operations typically appear in dependency chains, which might result in guidance to implementations or perhaps even modifications to the definition of memory_order_consume.

3.1 Types of Linux-Kernel Dependency Chains

One goal for memory_order_consume is to implement rcu_dereference(), which heads a Linux-kernel dependency-ordering tree. There are a number of variant of rcu_dereference() in the Linux kernel in order to implement the four flavors of RCU and also to enable RCU usage diagnositics for code that is shared by readers and updaters. These additional variants are rcu_dereference(), rcu_dereference_bh(), rcu_dereference_bh_check(), rcu_dereference_bh_check(), rcu_dereference_check(), rcu_dereference_index_check(), rcu_dereference_protected(), rcu_dereference_raw(), rcu_dereference_sched(), rcu_dereference_sched_check(), srcu_dereference(), and srcu_dereference_check(). Taken together, there are about 1300 uses of these functions in version 3.13 of the Linux kernel. However, about 250 of those are rcu_dereference_protected(), which is used only in update-side code and thus does not head up read-side dependency chains, which leaves about 1000 uses to be inspected for dependency-ordering usage.

3.2 Operators in Linux-Kernel Dependency Chains

A surprisingly small fraction of the possible C operators appear in dependency chains in the Linux kernel, namely ->, infix =, casts, prefix &, prefix *, [], infix +, infix -, ternary ?:, and infix (bitwise) &.

By far the two most common operators are the -> pointer field selector and the -> assignment operator. Enabling the carries-dependency relationship through only these two operators would likely cover better than 90% of the Linux-kernel use cases.

Casts, the prefix * indirection operator, and the prefix & address-of operator are used to implement Linux's list primitives, which translate from list pointers embedded in a structure to the structure itself. These operators are also used to get some of the effects of C++ subtyping in the C language.

The [] array-indexing operator, and the infix + and - arithmetic operators are used to manipulate RCU-protected arrays, as well as to index into arrays contained within RCU-protected structures. RCU-protected arrays are becoming less common because they are being converted into more complex data structures, such as trees. However, RCU-protected structures containing arrays are still fairly common.

The ternary ?: if-then-else expression is used to handle default values for RCU-protected pointers, for example, as shown in Figure 6, or in C++11 form in Figure 7. Note that the dependency is carried only through the rightmost two operands of ?:, never through the leftmost one.

The infix & operator is used to mask low-order bits from RCU pointers. These bits are used by some algorithms as markers. Such markers, though not common in the Linux kernel, are well-known in the art, with hazard pointers being but one example [25]. Note that it is expected that both operands of infix

```
struct foo {
  int a;
};
struct foo *fp;
struct foo default_foo;

int bar(void)
{
  struct foo *p;

  p = rcu_dereference(fp);
  return p ? p->a : default_foo.a;
}
```

Figure 6: Default Value For RCU-Protected Pointer, Linux Kernel

```
class foo {
  int a;
};
std::atomic<foo *> fp;
foo default_foo;

int bar(void)
{
  std::atomic<foo *> p;

  p = fp.load_explicit(memory_order_consume);
  return p ? kill_dependency(p->a) : default_foo.a;
}
```

Figure 7: Default Value For RCU-Protected Pointer, C++11

`&` are expected to have some non-zero bits, because otherwise a `NULL` pointer will result (at least in most implementations), and `NULL` pointers cannot reasonably be said to carry much of anything, let alone a dependency. Although I did not find any infix `|` operators in my census of Linux-kernel dependency chains, symmetry considerations argue for also including it, for example, for read-side pointer tagging. Presumably both of the operands of infix `|` must have at least one zero bit.

To recap, the operators appearing in Linux-kernel dependency chains are: `->`, infix `=`, casts, prefix `&`, prefix `*`, `[]`, infix `+`, infix `-`, ternary `?:`, infix (bitwise) `&`, and probably also `|`.

3.3 Operators Terminating Linux-Kernel Dependency Chains

Although C++11 has the `kill_dependency()` function to terminate a dependency chain, no such function exists in the Linux kernel. Instead, Linux-kernel dependency chains are judged to have terminated upon exit from the outermost RCU read-side critical section,[3] when existence guarantees are handed off from RCU to some other synchronization mechanism (usually locking or reference counting), or when the variable carrying the dependency goes out of scope.

That said, it is possible to analyze Linux-kernel dependency chains to see what part of the chain is actually required by the algorithm in question. We can therefore define the *essential subset* of a dependency chain to be that subset within which ordering is required by the algorithm. In the 3.13 version of the Linux kernel, the following operators always mark the end of the essential subset of a dependency chain: `()`, `!`, `==`, `!=`, `&&`, `||`, infix `*`, `/`, and `%`.

The postfix `()` function-invocation operator is an interesting special case in the Linux kernel. In theory, RCU could be used to protect JITed function bodies,

[3] The beginning of a given RCU read-side critical section is marked with `rcu_read_lock()`, `rcu_read_lock_bh()`, `rcu_read_lock_sched()`, or `srcu_read_lock()`, and the end by the corresponding primitive from the list `rcu_read_unlock()`, `rcu_read_unlock_bh()`, `rcu_read_unlock_sched()`, or `srcu_read_unlock()`. There is currently no C++11 counterpart for an RCU read-side critical section.

but in current practice RCU is instead used to wait for all pre-existing callers to the function referenced by the previous pointer. The functions are all compiled into the kernel, and the dependency chains are therefore irrelevant to the `()` operator. Hence, in version 3.13 of the Linux kernel, the `()` operator marks the end of the essential subset of any dependency chain that it resides in.

The `!`, `==`, `!=`, `&&`, and `||` operators are used exclusively in "if" statements to make control-flow decisions, and therefore also mark the end of the essential subset of any dependency chains that they reside in. In theory, these relational and boolean operators could be used to form array indexes, but in practice the Linux kernel does not yet do this in RCU dependency chains. The other relational operators (`>`, `<`, `>=`, and `<=`) should probably also be added to this list.

The infix `*`, `/`, and `%` arithmetic operators could potentially be used for construct array addresses, but they are not yet used that way in the Linux kernel. Instead, they are used to do computation on values fetched as the last operation in an essential subset of a dependency chain.

In short, in the current Linux kernel, `()`, `!`, `==`, `!=`, `&&`, `||`, infix `*`, `/`, and `%` all mark the end of the essential subset of a dependency chain. That said, there is potential for them to be used as part of the essential subset of dependency changes in future versions of the Linux kernel. And the same is of course true of the remaining C-language operators, which did not appear within any of the dependency chains in version 3.13 of the Linux kernel.

3.4 Linux-Kernel Dependency Chain Length

Many Linux-kernel dependency chains are very short and contained, with a fair number living within the confines of a single C statement. If there were only a few short dependency chains in the Linux kernel, one could imagine decorating all the operators in each dependency chain, for example, replacing the `->` operator with something like the mythical `field_dep()` operator shown on lines 16, 19, and 20 of Figure 8.

```
 1 void new_element(struct foo **pp, int a)
 2 {
 3   struct foo *p = malloc(sizeof(*p));
 4
 5   if (!p)
 6     abort();
 7   p->a = a;
 8   atomic_store_explicit(pp, p, memory_order_release);
 9 }
10
11 int traverse(struct foo_head *ph)
12 {
13   int a = -1;
14   struct foo *p;
15
16   p = atomic_load_explicit(&field_dep(ph, h),
17                            memory_order_consume);
18   while (p != NULL) {
19     a = field_dep(p, a);
20     p = atomic_load_explicit(&field_dep(p, n),
21                              memory_order_consume);
22   }
23   return a;
24 }
```

Figure 8: Decorated Linked Structure Traversal

However, there are a great many dependency chains that extend across multiple functions. One relatively modest example is in the Linux network stack, in the `arp_process()` function. This dependency chain extends as follows, with deeper nesting indicating deeper function-call levels:

- The `arp_process()` function invokes `__in_dev_get_rcu()`, which returns an RCU-protected pointer. The head of the dependency chain is therefore within the `__in_dev_get_rcu()` function.
- The `arp_process()` function invokes the following macros and functions:
  - `IN_DEV_ROUTE_LOCALNET()`, which expands to the `ipv4_devconf_get()` function.
  - `arp_ignore()`, which in turn calls:
    - `IN_DEV_ARP_IGNORE()`, which expands to the `ipv4_devconf_get()` function.
    - `inet_confirm_addr()`, which calls:
      - `dev_net()`, which in turn calls `read_pnet()`.
  - `IN_DEV_ARPFILTER()`, which expands to `ipv4_devconf_get()`.

  - `IN_DEV_CONF_GET()`, which also expands to `ipv4_devconf_get()`.
  - `arp_fwd_proxy()`, which calls:
    * `IN_DEV_PROXY_ARP()`, which expands to `ipv4_devconf_get()`.
    * `IN_DEV_MEDIUM_ID()`, which also expands to `ipv4_devconf_get()`.
  - `arp_fwd_pvlan()`, which calls:
    * `IN_DEV_PROXY_ARP_PVLAN()`, which expands to `ipv4_devconf_get()`.
  - `pneigh_enqueue()`.

Again, although a great many dependency chains in the Linux kernel are quite short, there are quite a few that spread both widely and deeply. We therefore cannot expect Linux kernel hackers to look fondly on any mechanism that requires them to decorate each and every operator in each and every dependency chain as was shown in Figure 8. In fact, even `kill_dependency()` will likely be an extremely difficult sell.

4 Dependency Ordering in Pre-C11 Implementations

Pre-C11 implementations of the C language do not have any formal notion of dependency ordering, but these implementations are nevertheless used to build the Linux kernel—and most likely all other software using RCU. This section lays out a few straightforward rules for both implementers (Section 4.2) and users of these pre-C11 C-language implementations (Section 4.1).

4.1 Rules for C-Language RCU Users

The primary rule for developers implementing RCU-based algorithms is to avoid letting the compiler determing the value of any variable in any dependency chain. This primary rule implies a number of secondary rules:

1. Use only intrinsic operators on basic types. If you are making use of C++ template metaprogramming or operator overloading, more elaborate rules apply, and those rules are outside the scope of this document.
2. Use a volatile load to head the dependency chain. This is necessary to avoid the compiler repeating the load or making use of (possibly erroneous) prior knowledge of the contents of the memory location, each of which can break dependency chains.
3. Avoid use of single-element RCU-protected arrays. The compiler is within its right to assume that the value of an index into such an array must necessarily evaluate to zero. The compiler could then substitute the constant zero for the computation, breaking the dependency chain and introducing misordering.
4. Avoid cancellation when using the `+` and `-` infix arithmetic operators. For example, for a given variable x, avoid $(x - x)$. The compiler is within its rights to substitute zero for any such cancellation, breaking the dependency chain and again introducing misordering. Similar arithmetic pitfalls must be avoided if the infix `*`, `/`, or `%` operators appear in the essential subset of a dependency chain.
5. Avoid all-zero operands to the bitwise `&` operator, and similarly avoid all-ones operands to the bitwise `|` operator. If the compiler is able to deduce the value of such operands, it is within its rights to substitute the corresponding constant for the bitwise operation. Once again, this breaks the dependency chain, introducing misordering.
6. If you are using RCU to protect JITed functions, so that the `()` function-invocation operator is a member of the essential subset of the dependency tree, 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.

7. Do not use the boolean `&&` and `||` operators in essential dependency chains. The reason for this prohibition is that they are often compiled using branches. Weak-memory machines such as ARM or PowerPC order stores after such branches, but can speculate loads, which can break data dependency chains.

8. Do not use relational operators (`==`, `!=`, `>`, `>=`, `<`, or `<=`) in the essential subset of a dependency chain. The reason for this prohibition is that, as for boolean operators, relational operators are often compiled using branches. Weak-memory machines such as ARM or PowerPC order stores after such branches, but can speculate loads, which can break dependency chains.

9. Be very careful about comparing pointers in the essential subset of a dependency chain. As Linus Torvalds explained, if the two pointers are equal, the compiler could substitute the pointer you are comparing against for the pointer in the essential subset of the dependency chain. On ARM and Power hardware, it might be that only the original value carried a hardware dependency, so this substitution would break the chain, in turn permitting misordering. Such comparisons are OK in the following cases:

   (a) The pointer being compared against references memory that was initialized at boot time, or otherwise long enough ago that readers cannot still have pre-initialized data cached. Examples include module-init time for module code, before kthread creation for code running in a kthread, while the update-side lock is held, and so on.

   (b) The pointer is never dereferenced after being compared. This exception occurs when comparing against the `NULL` pointer or when scanning RCU-protected circular linked lists.

   (c) The pointer being compared against is part of the essential subset of a dependency chain. This can be a different dependency chain, but *only* as long as that chain stems from a pointer that was modified after any initialization of interest. This exception can apply when carrying out RCU-protected traversals from different entry points that converged on the same data structure.

   (d) The pointer being compared against is fetched using `rcu_access_pointer()` and all subsequent dereferences are stores.

   (e) The pointers compared not-equal *and* the compiler does not have enough information to deduce the value of the pointer. (For example, if the compiler can see that the pointer will only ever take on one of two values, then it will be able to deduce the value based on a not-equals comparison.)

10. 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.

4.2 Rules for C-Language Implementers

The main rule for C-language implementers is to avoid any sort of value speculation, or, at the very least, provide means for the user to disable such speculation. An example of a value-speculation optimization that can be carried out with the help of hardware branch prediction is shown in Figure 9, which is an optimized version of the code in Figure 5. This sort of transformation might result from feedback-directed optimization, where profiling runs determined that the value loaded from `ph` was almost alway `0xbadfab1e`. Although this transformation is correct in a single-threaded environment, in a concurrent environment, nothing stops the compiler or the CPU from speculating the load on line 19 before it executes the `rcu_dereference()` on line 16, which could result in line 19 executing before the corresponding store on line 7, resulting in a garbage value in variable `a`.[4]

There *are* some situations where this sort of optimization would be safe, including:

[4] Kudos to Olivier Giroux for pointing out use of branch prediction to enable value speculation.

```
void new_element(struct foo **pp, int a)
{
  struct foo *p = malloc(sizeof(*p));

  if (!p)
    abort();
  p->a = a;
  rcu_assign_pointer(pp, p);
}

int traverse(struct foo_head *ph)
{
  int a = -1;
  struct foo *p;

  p = rcu_dereference(&ph->h);
  while (p != NULL) {
    if (p == (struct foo *)0xbadfab1e)
      a = ((struct foo *)0xbadfab1e)->a;
    else
      a = p->a;
    p = rcu_dereference(&p->n);
  }
  return a;
}
```

Figure 9: Dangerous Optimizations: Hardware Branch Predictions

1. The value speculated is a numeric value rather than a pointer, so that if the guess proves correct after the fact, the computation will be appropriate after the fact.
2. The value speculated is a pointer to invariant data, so that reasonable values are produced by dereeferencing, even if the guess proves to have been correct only after the fact.
3. As above, but where any updates result in data that produces appropriate computations at any and all phases of the update.

However, this list does not contain the general case of `memory_order_consume` loads.

Pure hardware implementations of value speculation can avoid this problem because they monitor cache-coherence protocol events that would result from some other CPU invalidating the guess.

In short, compiler writers must provide means to disable all forms of value speculation, unless the speculation is accompanied by some means of detecting the race condition that Figure 9 is subject to.

Are there other dependency-breaking optimizations that should be called out separately?

5 Dependency Ordering in C11 and C++11 Implementations

The simplest way to avoid dependency-ordering issues is to strengthen all `memory_order_consume` operations to `memory_order_acquire`. This functions correctly, but may result in unacceptable performance due to memory-barrier instructions on weakly ordered systems such as ARM and PowerPC,[5] and may further unnecessarily suppress code-motion optimizations.

Another straightforward approach is to avoid value speculation and other dependency-breaking optimizations. This might result in missed opportunities for optimization, but avoids any need for dependency-chain annotations and also all issues that might otherwise arise from use of dependency-breaking optimizations. This approach is fully compatible with the Linux kernel community's current approach to dependency chains. Unfortunately, there are any number of valuable optimizations that break dependency chains, so this approach seems impractical.

A third approach is to avoid value speculation and other dependency-breaking optimizations in any function containing either a `memory_order_consume` load or a `[[carries_dependency]]` attribute. For example, the hardware-branch-predition optimization shown in Figure 9 would be prohibited in such functions, as would cancellation optimizations such as optimizing `a = b + c - c` into `a = b`. This too can result in missed opportunities for optimization, though very probably many fewer than the previous approach. This approach can also result in issues due to dependency-breaking optimizations in functions lacking `[[carries_dependency]]` attributes, for example, function `d()` in Figure 10. It can also result in spurious memory-barrier instructions when a de-

[5] From a Linux-kernel community viewpoint, that should read "*will* result in unacceptable performance".

```
int a(struct foo *p [[carries_dependency]])
{
  return kill_dependency(p->a != 0);
}

int b(int x)
{
  return x;
}

foo *c(void)
{
  return fp.load_explicit(memory_order_consume);
  /* return rcu_dereference(fp) in Linux kernel. */
}

int d(void)
{
  int a;
  foo *p;

  rcu_read_lock();
  p = c();
  a = p->a;
  rcu_read_unlock();
  return a;
}
```

Figure 10: Example Functions for Dependency Ordering, Part 1

```
[[carries_dependency]] struct foo *e(void)
{
  return fp.load_explicit(memory_order_consume);
  /* return rcu_dereference(fp) in Linux kernel. */
}

int f(void)
{
  int a;
  foo *p;

  rcu_read_lock();
  p = e();
  a = p->a;
  rcu_read_unlock();
  return kill_dependency(a);
}

int g(void)
{
  int a;
  foo *p;

  rcu_read_lock();
  p = e();
  a = p->a;
  rcu_read_unlock();
  return b(a);
}
```

Figure 11: Example Functions for Dependency Ordering, Part 2

pendency chain goes out of scope, for example, with the `return` statement of function `g()` in Figure 11.

A fourth approach is to add a compile-time operation corresponding to the beginning and end of RCU read-side critical section. These would need to be evaluated at compile time, taking into account the fact that these critical sections can nest and can be conditionally entered and exited. Note that the exit from an outermost RCU read-side critical section should imply a `kill_dependency()` operation on each variable that is live at that point in the code.[6] Although it is probably impossible to precisely determine the bounds of a given RCU read-side critical section in the general case, conservative approaches that might overestimate the extent of a given section should be acceptable in almost all cases. This approach would make functions `c()` and `d()` in Figure 10 handle dependency chains in a natural manner, but avoiding whole-program analysis would require something similar to the `[[carries_dependency]]` annotations called out in the C11 and C++11 standards.

A fifth approach would be to require that all operations on the essential subset of any dependency chain be annotated. This would greatly ease implementation, but would not be likely to be accepted by the Linux kernel community.

A sixth approach is to track dependencies as called out in the C11 and C++11 standards. However, instead of emitting a memory-barrier instruction when a dependency chain flows into or out of a function without the benefit of `[[carries_dependency]]`, insert an implicit `kill_dependency()` invocation. Implementation should also optionally issue a diagnostic in this case. The motivation for this approach is that it is expected that many more `kill_dependencies()` than `[[carries_dependency]]` would be required to convert the Linux kernel's RCU code to C11. In the example in Figure 11, this approach would allow function `g()` to avoid emitting an unnecessary memory-barrier instruction, but without function `f()`'s explicit `kill_dependency()`. Both functions are in Fig-

[6] What if a given `rcu_read_unlock()` sometimes marked the end of an outermost RCU read-side critical section, but other times was nested in some other RCU read-side critical section? In that case, there should be no `kill_dependency()`.

```
1 p = atomic_load_explicit(gp, memory_order_consume);
2 if (p == ptr_a)
3   a = p->special_a;
4 else
5   a = p->normal_a;
```

Figure 12: Dependency-Ordering Value-Narrowing Hazard

ure 11.

A seventh and final approach is to track dependencies as called out in in the C11 and C++11 standards. With this approach, functions `e()` and `f()` properly preserve the required amount of dependency ordering.

6 Weaknesses in C11 and C++11 Dependency Ordering

Experience has shown several weaknesses in the dependency ordering specified in the C11 and C++11 standards:

1. The C11 standard does not provide attributes, and in particular, does not provide the `[[carries_dependency]]` attribute. This prevents the developer from specifying that a given dependency chain passes into or out of a given function.

2. The implementation complexity of the dependency-chain tracking required by both standard can be quite onerous on the one hand, and the overhead of unconditionally promoting `memory_order_consume` loads to `memory_order_acquire` can be excessive on weakly ordered implementations on the other. There is therefore no easy way out for a `memory_order_consume` implementation on a weakly ordered system.

3. The function-level granularity of `[[carries_dependency]]` seems too coarse. One problem is that points-to analysis is non-trivial, so that compilers are likely to have difficulty determining whether or not a given pointer carries a dependency. For example, the current wording of the standard (intentionally!) does not disallow dependency chaining through stores and loads. Therefore, if a dependency-carrying value might ever be written to a given variable, an implementation might reasonably assume that *any* load from that variable must be assumed to carry a dependency.

4. The rules set out in the standard [27, 1.10p9] do not align well with the rules that developers must currently adhere to in order to maintain dependency chains when using pre-C11 and pre-C++11 compilers (see Section 4.1). For example, the standard requires `x-x` to carry a dependency, and providing this guarantee would at the very least require the compiler to also turn off optimizations that remove `x-x` (and similar patterns) if `x` might possibly be carrying a dependency. For another example, consider the value-speculation-like code shown in Figure 12 that is sometimes written by developers, and that was described in bullet 9 of Section 4.1. In this example, the standard requires dependency ordering between the `memory_order_consume` load on line 1 and the subsequent dereference on line 3, but a typical compiler would not be expected to differentiate between these two apparently identical values. These two examples show that a compiler would need to detect and carefully handle these cases either by artificially inserting dependencies, omitting optimizations, differentiating between apparently identical values, or even by emitting `memory_order_acquire` fences.

5. The whole point of `memory_order_consume` and the resulting dependency chains is to allow developers to optimize their code. Such optimization attempts can be completely defeated by the `memory_order_acquire` fences that the standard currently requires when a dependency chain goes out of scope without the benefit of a `[[carries_dependency]]` attribute. Preventing the compiler from emitting these fences requires liberal use of `kill_dependency()`, which clutters code, requires large developer effort, and further requires that the developer know quite a bit about

which code patterns a given version of a given compiler can optimize (thus avoiding needless fences) and which it cannot (thus requiring manual insertion of `kill_dependency()`.

As of this writing, no known implementations fully support C11 or C++11 dependency ordering.

It is worth asking why Paul didn't anticipate these weaknesses. There are several reasons for this:

1. Compiler optimizations have become more aggressive over the seven years since Paul started working on standardization.
2. New dependency-ordering use cases have arisen during that same time, in particular, there are longer dependency chains and more of them, including dependency chains spanning multiple compilation units.
3. The number of dependency chains has increased by roughly an order of magnitude during that time, so that changes in code style can be expected to face a commeasurate increase in resistance from the Linux kernel community – unless those changes bring some tangible benefit.

With that, let's look at some potential alternatives to dependency ordering as defined in the C11 and C++11 standards.

7 Potential Alternatives to C11 and C++11 Dependency Ordering

Given the weaknesses in the current standard's specification of dependency ordering, it is quite reasonable to consider alternatives. To this end, Section 7.1 enlists help from the type system, but also imposes value restrictions (thus revising the C11 and C++11 semantics for dependencies), Section 7.2 enlists help from the type system without the value restrictions, Section 7.3 describes a whole-program approach to dependency chains (also revising the C11 and C++11 semantics for dependencies), and finally Section 7.4 discusses ease-of-use issues involved with revisions to the C11 and C++11 definitions of dependency ordering. Each approach appears to have advantages and disadvantages, so it is hoped that further discussion will either help settle on one of these alternatives or generate something better.

```
value_dep_preserving struct foo *p;

p = atomic_load_explicit(gp, memory_order_consume);
q = some_other_pointer;
if (p == q)
  do_something_with(p->a);
else
  do_something_else_with(p->b);
```

Figure 13: Single-Value Variables and Dependency Ordering

7.1 Type-Based Designation of Dependency Chains With Restrictions

This approach was formulated by Torvald Riegel in response to Linus Torvalds's spirited criticisms of the current C11 and C++11 wording.

This approach introduces a new `value_dep_preserving` type qualifier. Dependency ordering is preserved only via variables having this type qualifier. This is meant to model the real scope of dependencies, which is data flow, not execution at function-level granularity. This approach should therefore give developers much finer control of which dependencies are tracked.

Assigning from a `value_dep_preserving` value to a non-`value_dep_preserving` variable terminates the tracking of dependencies in much the same way that an explicit `kill_dependency()` would. However, unlike an explicit `kill_dependency()`, compilers should be able to emit a disablable warning on implicit conversions, so as to alert the developer about otherwise silent dropping of dependency tracking.[7]

Next, we specify that `memory_order_consume` loads return a `value_dep_preserving` type by default; the compiler must assume such a load to be capable of

[7] Other choices are possible in this case, including emitting a `memory_order_acquire` fence in order to conservatively preserve a potentially intended ordering.

producing any value of the underlying type. In other words, the implementation is not permitted to apply any value-restriction knowledge it might gain from whole-program analysis.

This allows developers to start with a clean slate for the additional rule that they must follow to be able to rely on dependency ordering: Subsequent execution must not lead to a situation there is only one possible value for the `value_dep_preserving` expression, because otherwise the implementation is permitted to break the dependency chain. This is shown in Figure 13, where the compiler is permitted to break dependency ordering on line 6 because it knows that the value of `p` is equal to that of `q`, which means that it could substitute the latter value from the former, which would break dependency ordering.

This approach has several advantages:

1. The implementation is simpler because no dependency chains need to be traced. The implementation can instead drive optimization decisions strictly from type information.

2. Use of the `value_dep_preserving` type modifier allows the developer to limit the extent of the dependency chains.

3. This type modifier can be used to mark a dependency chain's entry to and exit from a function in a straightforward way, without the need for attributes.

4. The `value_dep_preserving` type modifiers serve as valuable documentation of the developer's intent.

5. This approach permits many additional optimizations compared to those permitted by the current standard on code that carries a dependency. Expressions such as `x-x` no longer require establishment of artificial dependencies and the compiler is no longer required to detect value-narrowing hazards like that shown in Figure 12. However, the compiler is still prohibited from adding its own value-speculation optimizations.

6. Linus Torvalds seems to be OK with it, which indicates that this set of rules might be practical from the perspective of developers who currently exploit dependency chains.

According to Peter Sewell, one disadvantage is that this approach will be quite difficult to model, which in turn will pose obstacles for the analysis tooling that will be increasingly necessary for large-scale concurrent programming efforts. In particular, the concern is that forcing the compiler to assume that a `memory_order_consume` load could possibly return any value permitted by its type might require program-analysis tools to consider counterfactual hypothetical executions, which might complicate specification of semantics and verification.

7.2 Type-Based Designation of Dependency Chains

Jeff Preshing made an off-list suggestion of using a `value_dep_preserving` type modifier as suggested by Torvald Riegel, but using this type modifier to strictly enforce dependency ordering. For example, consider the code fragment shown in Figure 13. The scheme described in Section 7.1 would *not* necessarily enforce dependency ordering between the load on line 3 and the access one line 6, while the approach described in this section would enforce dependency ordering in this case.

Furthermore, cancelling or value-destruction operations on `value_dep_preserving` values would *not* disrupt dependency ordering. As with the current C11 and C++11 standards, the implementation would be required to emit a memory-barrier instruction or compute an artificial dependency for such operations. (Note however that use of cancelliong or value-destruction operations on dependency chains has proven quite rare in practice.)

This approach shares many of the advantages of Torvald Riegel's approach:

1. The implementation is simpler because no dependency chains need be traced. The implementation can instead drive optimization decisions strictly from type information.

2. Use of the `value_dep_preserving` type modifier allows the developer to limit the extent of the dependency chains.

3. This type modifier can be used to mark a dependency chain's entry to and exit from a function in a straightforward way, without the need for attributes.

4. The `value_dep_preserving` type modifiers serve as valuable documentation of the developer's intent.

5. Although optimizations on a dependency chain are restricted just as in the current standard, the use of `value_dep_preserving` restricts the dependency chains to those intended by the developer.

6. Restricting dependency-breaking optimizations on all dependency chains marked `value_dep_preserving`, without exceptions for cases in which the compiler knows too much, might make this approach easier to learn and to use.

It is expected that modeling this approach should be more straightforward.

7.3 Whole-Program Option

This approach, also suggested off-list by Jeff Preshing, has the goal of reusing existing non-dependency-ordered source code unchanged (albeit requiring recompilation in most cases).[8] For example, this approach permits an instance of `std::map` to be referenced by a pointer loaded via `memory_order_consume` and to provide that `std::map` instance with the benefits of dependency ordering without any code changes whatsoever to `std::map`. It is important to note that this protection will be provided only to a read-only `std::map` that is referenced by a changing pointer loaded via `memory_order_consume`, in particular, *not* to a concurrently updated `std::map` referenced by a pointer (read-only or otherwise) loaded via `memory_order_consume`. This latter case *would* require changes to the underlying `std:map` implementation, at a minimum, changing some of the loads to be `memory_order_consume` loads. Nevertheless, the ability to provide dependency-ordering protection to pre-existing linked data structures is valuable, even with this read-only restriction.

[8] A module or library that is known to never carry a dependency would not need to be recompiled.

This approach, which again does require full recompilation, can be implemented using two approaches:

1. Promote all `memory_order_consume` loads to `memory_order_acquire`, as may be done with the current standard.

2. Prohibit all dependency-breaking optimizations throughout the entire program, but only in cases where a change in the value returned by a `memory_order_consume` load could cause a change in the value computed later in that same dependency chain. Note again that the possibility of storing a value obtained from a `memory_order_consume` load, the loading it later, means that normal loads as well as `memory_order_relaxed` loads often must be considered to head their own dependency chains.

Some implementations might allow the developer to choose between these two approaches, for example, by using a compiler switch provided for that purpose.

This approach also has the effect of permitting a trivial implementation of a `memory_order_consume` `atomic_thread_fence()`. When using the first implementation approach, the `atomic_thread_fence()` is simply promoted to `memory_order_acquire`. Interestingly enough, when using the second approach, the `memory_order_consume` `atomic_thread_fence()` may simply be ignored. The reason for this is that this approach has the effect of promoting `memory_order_relaxed` loads to `memory_order_consume`, which already globally enforces all the ordering that the `memory_order_consume` `atomic_thread_fence()` is required to provide locally.

This approach has its own set of advantages and disadvantages:

1. This approach dispenses with the `[[carries_dependency]]` attribute and the `kill_dependency()` primitive.

2. This approach better promotes reuse of existing source code.

3. This approach allows implementations to carry out dependency-breaking optimizations on dependency chains as long as a change in the value from the `memory_order_consume` load does not change values further down the dependency chain, both with and without the optimization. Jeff conjectures that the set of dependency-breaking optimizations used in practice apply only outside of dependency chains, by the revised definition.[9] If this conjecture holds, it also applies to Torvald's approach described in Section 7.1.

4. Code that follows the rules presented in Section 4.1 (substituting `memory_order_consume` loads for `volatile` loads) would have its dependency ordering properly preserved.

7.4 Revising C11 and C++11 Dependency-Ordering Definition

The approaches described in Section 7.1 and 7.3 revise the dependency-ordering definition from that in the current C11 and C++11 standards.[10] In both cases, a dependency chain breaks if it comes to a point where only a single value is possible, regardless of the value of the `memory_order_consume` load heading up the chain. At first glance, this definition could be quite difficult to live with, as dependency ordering could come and go depending on small details of code far away from that point in the dependency chain.

However, a review of the Linux-kernel operators in Section 3.2 shows that the most commonly used operators act identically under both definitions. The problem-free operators include `->`, infix `=`, casts, prefix `&`, prefix `*`, and ternary `?:`.

One example of a potentially troublesome operator, namely `==`, was shown in Figure 13, where line 6 breaks dependency ordering because the value of `p` is known to be equal to that of `q`, which is not part of a dependency chain. This example could be addressed through careful diagnostic design coupled with appropriate coding standards. For example, the compiler could emit a warning on line 6, but remain silent for the equivalent line substituting `q` for `p`, namely, `do_something_with(q->a)`.

```
1 int my_array[MY_ARRAY_SIZE];
2
3 i = atomic_load_explicit(gi, memory_order_consume);
4 r1 = my_array[i];
```

Figure 14: Single-Element Arrays and Dependency Ordering

Another example is the use of postfix `[]` shown in Figure 14. If this code fragment was compiled with `MY_ARRAY_SIZE` equal to one, there is no dependency ordering between lines 3 and 4, but that same code fragment compiled with `MY_ARRAY_SIZE` equal to two or greater *would* be dependency-ordered. Here a diagnostic for single-element arrays might prove useful.

In the Linux kernel, infix `+` and `-` are used for pointer and array computations. These are all safe in that they operate on an integer and pointer, so that any cancellation will not normally be detectable at compile time. However, one big purpose of diagnostics is to detect abnormal conditions indicating probable bugs. Therefore, in cases where the compiler can determine that two values from dependency chains are annihilating each other via infix `+` and `-`, a diagnostic would be appropriate.

Similarly, the Linux kernel uses infix (bitwise) `&` to manipulate bits at the bottom of a pointer, where again cancellation will not normally be detectable at compile time—except in the case of operations on a `NULL` pointer, for which dependency ordering is not meaningful in any case. However, as wtih infix `+` and `-`, if the compiler detects value annihilation, a diagnostic would be appropriate.

Although issues with false positives and negatives needs further investigation, there is reason to hope that this revision of the definition of dependency ordering might avoid significant impacts on ease of use.

[9] This is certainly the case for the usual optimizations exemplified by replacing `x-x` with zero.

[10] The approach described in Section 7.2 provides clear syntactic delineation of what is and is not part of a dependency chain, so is not discussed further here.

8 Summary

This document has analyzed Linux-kernel use of dependency ordering and has laid out the status-quo interaction between the Linux kernel and pre-C11 compilers. It has also put forward some possible ways of building towards a full implementation of C11's and C++11's handling of dependency ordering. Finally, it calls out some weaknesses in C11's and C++11's handling of dependency ordering and offers some alternatives.

References

[1] Alglave, J., Maranget, L., Pawan, P., Sarkar, S., Sewell, P., Williams, D., and Nardelli, F. Z. PPCMEM/ARMMEM: A tool for exploring the POWER and ARM memory models. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/pldi105-sarkar.pdf`, June 2011.

[2] Alglave, J., Maranget, L., and Tautschnig, M. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 40–40.

[3] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2010.

[4] Boehm, H. J. Space efficient conservative garbage collection. *SIGPLAN Not. 39*, 4 (Apr. 2004), 490–501.

[5] Bonzini, P., and Day, M. RCU implementation for Qemu. `http://lists.gnu.org/archive/html/qemu-devel/2013-08/msg02055.html`, August 2013.

[6] Dalton, M. *THE DESIGN AND IMPLEMENTATION OF DYNAMIC INFORMATION FLOW TRACKING SYSTEMS FOR SOFTWARE SECURITY.* PhD thesis, Stanford University, 2009. Available: `http://csl.stanford.edu/~christos/publications/2009.michael_dalton.phd_thesis.pdf` [Viewed March 9, 2010].

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

[8] 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.

[9] Grisenthwaite, R. *ARM Barrier Litmus Tests and Cookbook.* ARM Limited, 2009.

[10] Howard, P. W., and Walpole, J. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2011), HotPar'11, USENIX Association, pp. 1–6.

[11] Howard, P. W., and Walpole, J. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience* (2013), n/a–n/a.

[12] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, 2002. Available: `http://developer.intel.com/design/itanium/downloads/251429.htm` `ftp://download.intel.com/design/Itanium/Downloads/25142901.pdf` [Viewed: January 10, 2007].

[13] International Business Machines Corporation. *Power ISA Version 2.07*, 2013.

[14] Kannan, H. Ordering decoupled metadata accesses in multiprocessors. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), ACM, pp. 381–390.

[15] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine,

part i. *Commun. ACM 3*, 4 (Apr. 1960), 184–195.

[16] McKenney, P. E. Read-copy update (RCU) usage in Linux kernel. Available: `http://www.rdrop.com/users/paulmck/RCU/linuxusage/rculocktab.html` [Viewed January 14, 2007], October 2006.

[17] McKenney, P. E. What is RCU? part 2: Usage. Available: `http://lwn.net/Articles/263130/` [Viewed January 4, 2008], January 2008.

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

[19] McKenney, P. E. Validating memory barriers and atomic instructions. `http://lwn.net/Articles/470681/`, December 2011.

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

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

[22] 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.

[23] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

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

[25] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (June 2004), 491–504.

[26] 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].

[27] Toit, S. D. Working draft, standard for programming language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3691.pdf`, May 2013.

[28] Vigueras, G., Orduña, J. M., and Lozano, M. A Read-Copy Update based parallel server for distributed crowd simulations. *The Journal of Supercomputing* (Apr. 2012).