N3934: Towards Implementation and Use of `memory_order_consume`

Doc. No.: WG21/N3934
Date: 2014-02-14
Reply to: Paul E. McKenney
Other contributors: Olivier Giroux, Lawrence Crowl, ...
Email: paulmck@linux.vnet.ibm.com

February 16, 2014

1 Introduction

Although RCU is gaining significant use both within and outside of the Linux kernel, there are no know 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.

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.

C11/C++11 `memory_order_consume` is intended to implement RCU's lightweight subscribe operation.

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 [2, 5, 6, 7, 8, 9, 10].

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

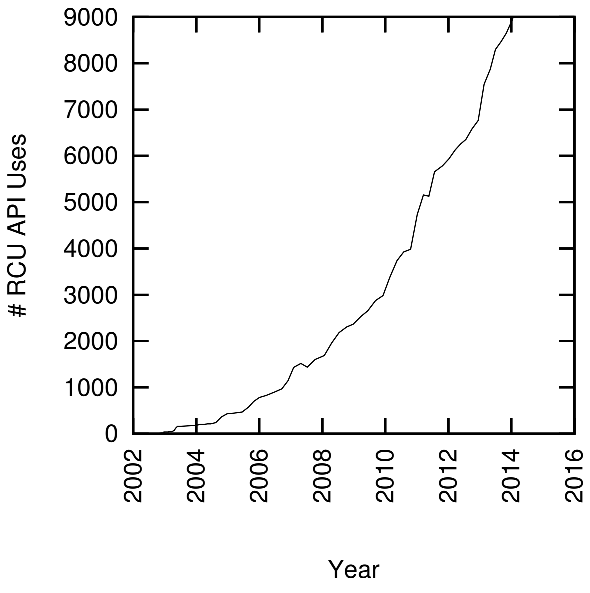

Figure 1: Growth of RCU Usage

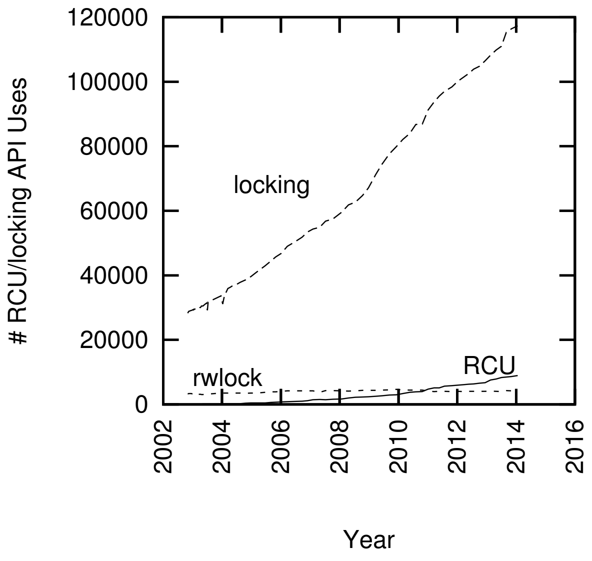

Figure 2: Growth of RCU Usage vs. Locking

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 [3, 4, 12].

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 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 [1]. 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.

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

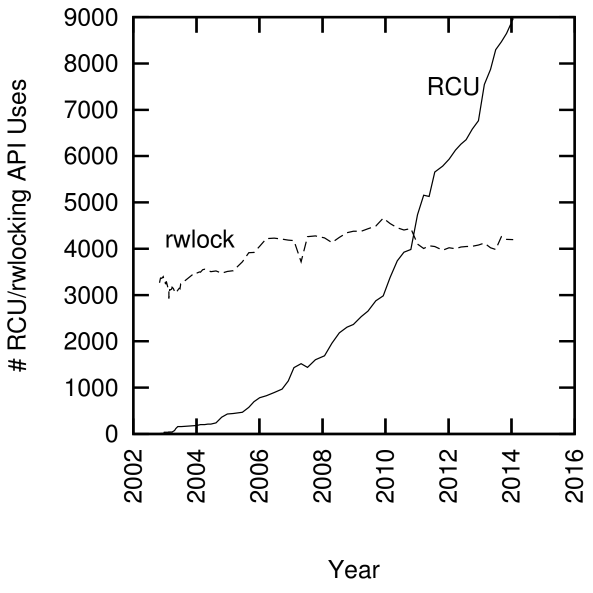

Figure 3: Growth of RCU Usage vs. Reader-Writer Locking

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.

```
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 4: Default Value For RCU-Protected Pointer, Linux Kernel

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 4, or in C++11 form in Figure 5. Note that the dependency is carried only through the rightmost two operands of `?:`, never through the leftmost one.

```
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 5: Default Value For RCU-Protected Pointer, C++11

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 [11]. Note that it is expected that both operands of infix `&` 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,[1] 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, 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

[1] 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.

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. 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. In fact, even use of `kill_dependency()` throughout 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 these 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. Similar rules apply to the boolean `&&` and `||` operators, should you use them in dependency chains such that the resulting value is used to determine the address of RCU-protected data. In addition, use of `&&` and `||` in essential subsets of dependency chains is hazardous because 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 dependency chains.

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. If relational operators (`==`, `!=`, `>`, `>=`, `<`, or `<=`) appear in the essential subset of a dependency chain, avoid situations where the compiler can guess the result, for example, based on prior knowledge of the signs of the operands. In addition, use of relational operators in essential subsets of dependency chains can be hazardous because, like the boolean operators, 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 dependency chains.

8. 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.

Classic value speculation can fall prey to the following sequence of events when applied to the essential subset of a dependency chain:

1. The compiler incorrectly guesses the value of a pointer `p`.

2. The compiler carries out accesses using this guess, and these accesses therefore return garbage.

3. Some other thread updates the value of pointer `p`, and the new value just happens to match the guess.

4. The compiler loads the actual value of pointer `p`, notes that this value matches the guess, and incorrectly concludes that the guess is correct. The aforementioned garbage is therefore passed into the program. Garbage in, garbage out!

Hardware avoids this problem because it monitors cache-coherence protocol events that would result from some other CPU invalidating the guess.

Interestingly enough, the compiler can use the branch predictor to gain access to these hardware cache-coherence events. For example, the compiler could compare its guess against a load from pointer `p`. If this guess had tended to be correct in the recent past, the branch predictor would cause the CPU to start speculatively executing the code in the `then` clause of the `if` statement. If some other CPU did a store, this speculative execution would be squashed.

Unfortunately, if the speculation succeeds, the code has used a constant unconnected to the actual pointer to access the data. There is therefore no dependency on the value loaded, and in turn no dependency ordering. Although the hardware won't speculate the stores, the CPU is within its rights to return old garbage values for the loads.[2]

In short, implementers must provide means to disable all forms of value speculation.

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

```
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 6: Example Functions for Dependency Ordering, Part 1

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,[3] 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.

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. 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 6. It can also result in spurious memory-barrier instructions when a dependency chain goes out of scope, for example, with the `return` statement of function `g()` in Figure 7.

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

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

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

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

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

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

Figure 7: Example Functions for Dependency Ordering, Part 2

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.[4] 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 6 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. This approach would allow function `g()` avoid emitting an unnecessary memory-barrier instruction, but without function `f()`'s explicit `kill_dependency()`. Both functions are in Figure 7.

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 Summary

This document has analyzed Linux-kernel use of dependency ordering and has laid out the status-quo interaction between the Linux kernel and compilers. It has also put forward some possible ways forward building on C11's and C++11's handling of dependency ordering, many of which will unfortunately be quite unpalatable to the Linux kernel community.

References

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

[2] Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update.

[4] 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()`.

IEEE Transactions on Parallel and Distributed Systems 23 (2012), 375–382.

[3] 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.

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

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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].