Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel

Jade Alglave
University College London
Microsoft Research
j.alglave@ucl.ac.uk

Luc Maranget
Inria — Paris
luc.maranget@inria.fr

Paul E. McKenney
IBM Corporation
Oregon State University
paulmck@linux.vnet.ibm.com

Andrea Parri
Scuola Superiore Sant'Anna
andrea.parri@sssup.it

Alan Stern
Harvard University
stern@rowland.harvard.edu

Abstract

Concurrency in the Linux kernel can be a contentious topic. The Linux kernel mailing list features numerous discussions related to consistency models, including those of the more than 30 CPU architectures supported by the kernel and that of the kernel itself. How are Linux programs supposed to behave? Do they behave correctly on exotic hardware?

A formal model can help address such questions. Better yet, an executable model allows programmers to experiment with the model to develop their intuition. Thus we offer a model written in the cat language, making it not only formal, but also executable by the herd simulator. We tested our model against hardware and refined it in consultation with maintainers. Finally, we formalised the *fundamental law of the Read-Copy-Update synchronisation mechanism*, and proved that one of its implementations satisfies this law.

ACM Reference Format:
Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3177156

1 Introduction

Concurrency in Linux may frighten small children [35]; it also appears to be disconcerting to grown-ups.

1.1 "Still confusion situation all round" [sic] [89]

The Linux kernel (LK) targets more than 30 CPU architectures, amongst which Alpha [18], ARM [14], IBM Power [38], Intel [40], Itanium [40] and MIPS [39] implement *weak consistency models*. Consistency models determine what values a read can take; weak models allow more behaviours than Sequential Consistency (SC) [45].

Table 1. Selection of LKML discussions

Model	URL
SPARC	[51, 76]
LK	[84]
LK	[4, 56]
LK	[19]
LK	[73]
LK	[36]
LK/Itanium	[16, 70]
LK	[20]
Itanium	[48, 49, 57, 88]
Intel	[41, 53]
LK/C11	[22, 23]
LK	[21]
Alpha	[79]
LK	[31]
ARM64	[26]
LK	[27]
MIPS	[81, 85–87]
Power	[30, 32, 33]
ARM64	[28]
LK/C11	[24]
LK	[72]

These architectures implement distinct models and thus disagree on the values that a read can return. This leads to a plethora of discussions on the Linux Kernel mailing list (LKML), some of which are listed in Table 1; their frequency has increased as multicore systems have gone mainstream.

LK developers must understand not only the kernel's concurrency primitives, but also those of the underlying hardware. Several documents make laudable efforts in this direction: [37] lists what orderings are guaranteed; [69] summarises semantics of read-modify-write operations, and [55] documents ways of avoiding counterproductive optimisations. Sadly these documents are in prose, subject to ambiguities and misinterpretations. As a candid disclaimer puts it [37]:

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

This document is not a specification; it is intentionally (for the sake of brevity) and unintentionally (due to being human) incomplete. [...] in case of any doubt (and there are many) please ask.

This quote suggests that a specification might dispel all doubts. However, as Linus Torvalds writes [78]:

*With specs, there really *are* people who spend years discussing what the meaning of the word "access" is or similar [the authors of this paper plead guilty]. Combine that with a big spec that is 500+ pages in size and then try to apply that all to a project that is 15 million lines of code and sometimes *knowingly* has to do things that it simply knows are outside the spec [...]"*

This highlights the need for an object beyond a prose specification: unambiguous, concise, amenable to vast code projects, and complete. We offer a *formal executable model* for the LK, written in the cat language [12].

Writing a memory consistency model in cat gives it a formal meaning, since cat has a formal semantics [3]. Moreover, a cat model can be executed within the herd tool [5], allowing users to experiment with the model to develop their intuition.

1.2 "[I]t is your kernel, so what is your preference?" [54]

Architects and standard committees are often seen as ultimate authorities on consistency matters. In our case, we rely on Linus Torvalds's and his maintainers' posts to LKML and the gcc mailing list. We cite and discuss these posts below.

A common denominator of hardware models seems to align with Torvalds' view [80]:

Weak memory ordering is [...] hard to think about [...] So the memory ordering rules should [...] absolutely be as tight as at all humanly possible, given real hardware constraints.

To this end, we axiomatised models of IBM Power [74, 75] in cat. We modified this formalisation to handle Alpha [18] and incorporate ideas from academic ARM models [34]. ARM then released their official memory model [47, Chap. B2.3] (including a cat file distributed within the diy+herd toolsuite [5]), making those models obsolete; we thus modified our LK model accordingly. This experience shows that our model will change over time as existing hardware evolves, or new hardware arises.

Yet the LK cannot simply be an envelope for the architectures it supports. As Ingo Molnar writes [71]:

it's not true that Linux has to offer a barrier and locking model that panders to the weakest (and craziest!) memory ordering model amongst all the possible Linux platforms—theoretical or real metal. Instead what we want to do is to consciously, intelligently pick a sane, maintainable memory model and offer primitives for that—at least as far as generic code is concerned. Each architecture can map those primitives to the best of its abilities.

This seems much like defining a language-level model: it might appear that the C11 model could be used as the LK model. Indeed, converging with C11 is the topic of several LKML discussions [22, 24]. Unfortunately the C11 model is an imperfect fit [78]:

I do not believe for a second that we can actually use the C11 memory model in the kernel [...] We will continue to have to do things that are "outside the specs" [...] with models that C11 simply doesn't cover.

In short, the LK should have a model of its own.

1.3 "[P]ick a sane, maintainable memory model" [71]

Our LK model is a first attempt at fulfilling this wish. Of course, concerns like sanity or maintainability are to an extent in the eye of the beholder. But we believe that the LK community will help achieve these goals. Indeed, our work is based on interactions with the community, along with documentation and posts to mailing lists. This has been necessary for understanding the semantics of certain pieces of code.

Table 2. LK issues that our work helped address

LK issue	URL
locking on ARM64	[26]
ambiguities in [37]	[59]
ambiguities in RCU documentation	[58]
CPU hotplug	[90]
assumption about lock-unlock	[64]
semantics of `spin_unlock_wait`	[83]

Moreover, our model has already resolved ambiguities and helped fix bugs (see Table 2). The RCU documentation now uses our definitions [58] and `memory-barriers.txt` [37] was updated to distinguish between transitivity and cumulativity [59]. Our work informed fixes to code incorrectly relying on fully ordered lock-unlock pairs [64], code where ARM64 needed stronger ordering from combinations of locking and fences [26], and discussions about the semantics of locking primitives [83]. Finally, our model was directly used by a maintainer to justify his patch [90]; this highlights the practical applicability of our model.

Seven maintainers agreed to sponsor our model, which has received positive feedback on LKML [60].

1.4 Correctness of concurrent code

Our model is also a stepping stone towards assessing the correctness of LK code. We focus here on *Read-Copy-Update* (RCU) [52].

CBMC [17] has been used to verify LK Tree RCU over SC, TSO, and PSO [46]; others used Nidhugg over SC and TSO [42]. Userspace RCU has been examined with respect to C11 [43, 77]. These works provided valuable insights, but only relative to the models available to them. We examine RCU in the light of our LK model, the first to provide a formal semantics for RCU. Moreover, our results provide two alternative ways to integrate a semantics of RCU in a software analysis tool.

1.5 Overview of the paper and contributions

Section 2 introduces LK programs and their executions, and the cat language. Section 3 describes and illustrates our model. Section 4 formalises RCU. Section 5 gives our experimental results. Section 6 examines the correctness of an RCU implementation. In summary, this paper presents:

1. a formal core LK memory model, in the form of a specification of the model in cat (Figure 8) and precise constraints under which executions are allowed or forbidden by the LK model (Figure 3);
2. examples illustrating how forbidden executions violate the constraints (Figures 2, 4, 5, 6, and 7);
3. a formalisation of RCU as an axiom (Figure 12);
4. a formalisation of the *fundamental law of RCU* [62], equivalent to the axiom (Theorem 1);
5. experiments showing that our model is sound with respect to hardware, and a comparison with C11 (Table 5);
6. the correctness of an RCU implementation (Theorem 2);
7. a discussion of required future work (Section 7).

The cat model, test results and proofs are online [7].

2 Programs and Candidate Executions

LK programs communicate via shared locations (e.g., `x`, `y`, `z`), use private locations (e.g., `r1`, `r2`) for logic or arithmetic, and control their execution flow with conditionals and loops. Use of shared accesses may result in weak behaviours.

Figure 1 shows an LK program where two threads communicate via shared locations `x` and `y`, initialised to 0. T_0 updates `x`, calls `smp_wmb`, and sets `y` to 1. T_1 reads `y`, calls `smp_rmb`, and reads `x`. This is a *message passing* idiom: with enough synchronisation, after T_1 sees that the flag `y` is set, it must see the updated data. Here `smp_wmb` and `smp_rmb` are enough.

```
int x=0, y=0;

void T0() {
 WRITE_ONCE(x,1);
 smp_wmb();
 WRITE_ONCE(y,1);
}

void T1() {
 int r1 = READ_ONCE(y);
 smp_rmb();
 int r2 = READ_ONCE(x);
}
```

Figure 1. LK program

Below we partially describe the LK primitives in Table 3, formalised in Figure 8. Table 4 details RCU primitives.

`ONCE` ***primitives*** are special reads and writes which restrict compiler optimisations (vide infra).

`Acquire` ***and*** `release` ***primitives*** are synchronising: a release read by an acquire ensures that writes before the release are seen by the acquire's thread.

Fences prevent reorderings: `smp_rmb` for reads, `smp_wmb` for writes, `smp_mb` for all accesses, and `smp_read_barrier_depends` for dependent reads on architectures that do not respect such dependencies, viz, Alpha.

Read-modify-writes (`xchg` and siblings) consist of a read paired with a write. Depending on the primitive, these reads and writes can be `ONCE` (for `xchg_relaxed`), `acquire`, `release`, or surrounded by full fences (for `xchg`).

Table 3. LK primitives and corresponding events

LK/C primitive	Event
`READ_ONCE()`	`R[once]`
`WRITE_ONCE()`	`W[once]`
`smp_load_acquire()`	`R[acquire]`
`smp_store_release()`	`W[release]`
`smp_rmb()`	`F[rmb]`
`smp_wmb()`	`F[wmb]`
`smp_mb()`	`F[mb]`
`smp_read_barrier_depends()`	`F[rb-dep]`
`xchg_relaxed()`	`R[once]`, `W[once]`
`xchg_acquire()`	`R[acquire]`, `W[once]`
`xchg_release()`	`R[once]`, `W[release]`
`xchg()`	`F[mb]`, `R[once]`, `W[once]`, `F[mb]`

LK coding conventions restrict compiler optimisations, e.g.:

- `ONCE` primitives prevent tearing (compiling a large access as a group of smaller accesses), fusing (compiling a series of accesses to a single location as just one access), and splitting (compiling a single access as multiple full-sized accesses, e.g., repeating a load to avoid a register spill);
- dependencies are crafted to prevent the compiler from breaking them [37, 55];
- the LK relies on inline assembly: for example, architectures with write memory barriers can implement `smp_wmb`, despite lack of C11 support for this notion.

In addition, we only model architectures that the LK actually supports. Thus we need not consider (for example) difficulties such as 8-bit architectures with 16-bit pointers. All in all, our LK model specifies the cumulative effect of a language-level model (the subset of C specific to the LK) and the hardware models targeted by the LK.

A consistency model determines which values can be returned by read primitives. An *axiomatic* model—the style we chose here—does so by determining whether *candidate executions* of a program are allowed. Candidate executions are graphs: nodes are *events* modeling instructions, and edges form *relations over events*, representing, e.g., the program order in which instructions appear on a thread, or where a read takes its value from. Figure 2 shows a candidate execution (with initial writes and thread labels omitted).

Events model primitives. Reads (R) from a shared location place the value read in a private location, writes (W) to a shared location update said location with a given value, and

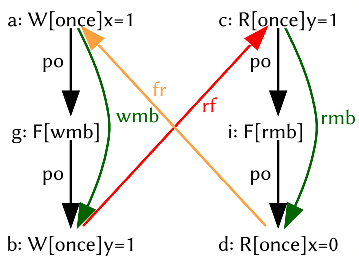

Figure 2. Forbidden execution for the program in Figure 1

fences (F) may prevent undesirable behaviours. Read-modify-writes give rise to a read and a write for the same shared location. Events bear *annotations* reflecting the corresponding primitives: `once` or `acquire` (for reads); `once` or `release` (for writes); and `rmb`, `wmb`, `mb` or `rb-dep` (for fences). For example, `smp_load_acquire` is represented by a read annotated `acquire`, `WRITE_ONCE` by a write annotated `once`, and `smp_wmb` by a fence annotated `wmb`. Table 3 lists the events for each primitive, omitting locations for brevity.

Candidate executions consist of *abstract executions*, representing the semantics of each thread, and *execution witnesses*, representing communications between threads. Abstract executions (E, `po`, `addr`, `data`, `ctrl`, `rmw`) contain:

- E, the set of events;
- `po`, the program order, specifies instruction order in a thread after evaluating conditionals and unrolling loops;
- `addr`, `data`, and `ctrl` are the *address*, *data*, and *control dependency* relations in `po`, always starting from a read.
- `rmw` links the read of a read-modify-write to its write.

Execution witnesses (`rf`, `co`) contain:

- the *reads-from* relation `rf`, which determines where reads take their value from. For each read r there is a unique write w to the same location s.t. r takes its value from w.
- the *coherence order* relation `co`, representing the history of writes to each location. It is a total order over writes to the same location, starting with the initialising write.

The cat language [3] formalises consistency models as sets of constraints over candidate executions. We wrote our model in cat so that it can be executed by the herd simulator [12]. The language provides the user with predefined sets of events (`W` contains all write events, `R` all reads, and `_` all events) and the relations forming candidate executions (`po`, `addr`, `data`, `ctrl`, `rmw`, `rf`, and `co`), as well as the identity relation `id`, the `loc` relation, which contains all pairs of events that access the same shared location, and the `int` relation, which contains all pairs of events that belong to the same thread.

Users can build new relations via union ($\cup$), intersection ($\cap$), difference ($\setminus$), complement ($\sim r$), inverse (r^{-1}), reflexive closure ($r^?$), transitive closure (r^+), reflexive transitive closure (r^*), sequence ($r_1 ; r_2$, defined as $\{(x,z) \mid \exists y[(x,y) \in r_1 \wedge (y,z) \in r_2]\}$), and direct product of sets of events ($X \times Y$).

One can thus build the following relations, which often appear in cat models (and in our LK model):

- the *from-reads* relation consists of one step of reads-from backwards, then one step of coherence: $\texttt{fr} := \texttt{rf}^{-1} ; \texttt{co}$;
- the *communication* relation gathers reads-from, coherence and from-reads: $\texttt{com} := \texttt{rf} \cup \texttt{co} \cup \texttt{fr}$;
- the *dependency* relation gathers address and data (but not control) dependencies: $\texttt{dep} := \texttt{addr} \cup \texttt{data}$;
- the program order relation *restricted to accesses of the same location*: $\texttt{po-loc} := \texttt{po} \cap \texttt{loc}$;
- the *internal reads-from* relation, i.e., the reads-from which take place within a thread: $\texttt{rfi} := \texttt{rf} \cap \texttt{int}$;
- the *external* relation `ext`, containing pairs of events that belong to different threads: $\texttt{ext} := \sim \texttt{int}$;
- the *external reads-from, coherence and from-reads*: $\texttt{rfe} := \texttt{rf} \cap \texttt{ext}$, $\texttt{coe} := \texttt{co} \cap \texttt{ext}$, and $\texttt{fre} := \texttt{fr} \cap \texttt{ext}$.

A cat model can constrain a relation r to be irreflexive, acyclic, or empty.

In Figure 2, read `c` takes its value from write `b`, hence the reads-from (`rf`) arrow between them. Read `d` takes the initial value, which is overwritten by write `a`, hence the from-reads (`fr`) arrow between them. This candidate execution is forbidden by our LK model: the synchronisation ensures that the updated data `x` is visible to T_1 when it reads the flag `y`.

3 The LK model's core

A candidate execution is *allowed by the core LK model* iff it satisfies the constraints of Figure 3; it is *forbidden* otherwise.

$$\text{acyclic}(\texttt{po-loc} \cup \texttt{com}) \qquad (\texttt{Scpv})$$
$$\text{empty}(\texttt{rmw} \cap (\texttt{fre} ; \texttt{coe})) \qquad (\texttt{At})$$
$$\text{acyclic}(\texttt{hb}) \qquad (\texttt{Hb})$$
$$\text{acyclic}(\texttt{pb}) \qquad (\texttt{Pb})$$

Figure 3. Core of our LK model

Constraint `Scpv` (*sequential consistency per variable*) forces the values of a single variable to be the ones it would have in SC: weak consistency arises from interactions among variables. `At` (*atomicity*) ensures that there cannot be an intervening write to the same location between the read and the write of a read-modify-write. `Hb` (*happens-before*) provides the intuitive causality notion. `Pb` (*propagates-before*) constrains the propagation of writes and fences among concurrent threads. Both `Scpv` and `At` appear in the literature [12, Sect. 4.2]. In this section, we present `Hb` and `Pb`.

These axioms constrain the `hb` and `pb` relations (defined later) to be partial orders, because they require the relations to be acyclic. Below we illustrate these orders using examples from the LK 4.12 source code [82].

Auxiliary relations in the figures include the following: The `acq-po` relation contains pairs of events in program order such that the first is an acquire. Similarly, `po-rel` pairs events where the second is a release. `rfi-rel-acq` is an internal reads-from communication between a release and an acquire. The `rmb` relation pairs reads with an `smp_rmb` fence between them. Similarly, `wmb` pairs writes with an `smp_wmb` fence between them, `mb` pairs any events with an `smp_mb` fence between them, and `rb-dep` pairs reads with a `smp_read_barrier_depends` between them.

3.1 Examples

LB+ctrl+mb, in Figure 4, appears in the ring-buffer interface from kernel to userspace (see `perf_output_put_handle()` in [82, `kernel/events/ring_buffer.c`]). T_0 reads from `x` (event a) and writes to `y` (b), imposing a control dependency (depicted by the `ctrl` arrow) in between. Similarly, T_1 reads from `y` (c) and writes to `x` (d), with an `smp_mb` fence between them (`mb` arrow). If the dependency or the fence is removed, the execution is allowed by the model and observed on ARMv7 [50, Sect. 7.1].

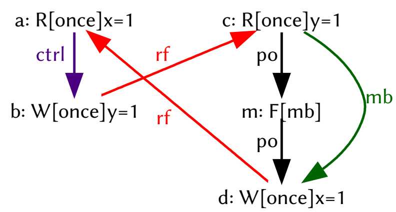

Figure 4. LB+ctrl+mb: Forbidden.

WRC+po-rel+rmb, in Figure 5, is a sibling of Figure 1. This pattern appears in LKML discussions [61]. T_0 writes to `x` (a), and T_1 writes to `y` (c) after reading `x` (b). The release in T_1 (`po-rel` arrow) forces a to happen before c, even though a and c are not in the same thread. The fence in T_2 (`rmb` arrow) ensures that d and e stay in order.

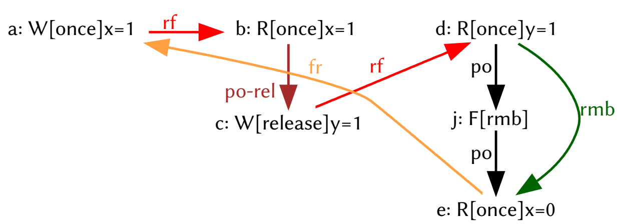

Figure 5. WRC+po-rel+rmb: Forbidden.

SB+mbs (a store buffering idiom), in Figure 6, is used in LK wait-event/wakeup code. It is documented in functions `waitqueue_active()` [82, `include/linux/wait.h`]; `wait_woken()` and `woken_wake_function()` [82, `kernel/sched/wait.c`]; and `wake_q_add()`, `wake_up_q()`, and `try_to_wake_up()` [82, `kernel/sched/core.c`]. Without the fences it is observed on x86.

PeterZ, in Figure 7, is used to resolve races between performance monitoring and CPU hotplug operations [90]. As in the previous example, two strong fences forbid the pattern, which otherwise is observed on Power machines.

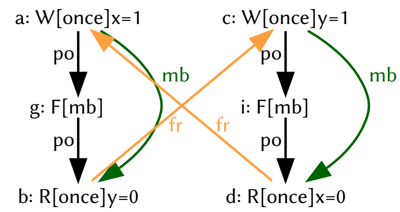

Figure 6. SB+mbs: Forbidden.

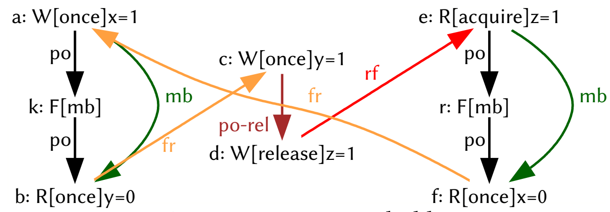

Figure 7. PeterZ: Forbidden.

3.2 Formal definitions

We now dive into the formal definitions of our model, given in Figure 8, which we justify in the light of the LK design.

$$
\begin{array}{l}
\mathtt{dep} := \mathtt{addr} \cup \mathtt{data} \\
\mathtt{rwdep} := (\mathtt{dep} \cup \mathtt{ctrl}) \cap (\mathtt{R} \times \mathtt{W}) \\
\mathtt{overwrite} := \mathtt{co} \cup \mathtt{fr} \\
\mathtt{to\text{-}w} := \mathtt{rwdep} \cup (\mathtt{overwrite} \cap \mathtt{int}) \\
\mathtt{rrdep} := \mathtt{addr} \cup (\mathtt{dep} \,;\, \mathtt{rfi}) \\
\mathtt{strong\text{-}rrdep} := \mathtt{rrdep}^{+} \cap \mathtt{rb\text{-}dep} \\
\mathtt{to\text{-}r} := \mathtt{strong\text{-}rrdep} \cup \mathtt{rfi\text{-}rel\text{-}acq} \\
\mathtt{strong\text{-}fence} := \mathtt{mb} \\
\mathtt{fence} := \mathtt{strong\text{-}fence} \cup \mathtt{po\text{-}rel} \cup \mathtt{wmb} \cup \mathtt{rmb} \cup \mathtt{acq\text{-}po} \\
\mathtt{ppo} := \mathtt{rrdep}^{*} \,;\, (\mathtt{to\text{-}r} \cup \mathtt{to\text{-}w} \cup \mathtt{fence}) \\
\mathtt{cumul\text{-}fence} := \mathtt{A\text{-}cumul}(\mathtt{strong\text{-}fence} \cup \mathtt{po\text{-}rel}) \cup \mathtt{wmb} \\
\mathtt{prop} := (\mathtt{overwrite} \cap \mathtt{ext})^{?} \,;\, \mathtt{cumul\text{-}fence}^{*} \,;\, \mathtt{rfe}^{?} \\
\mathtt{hb} := ((\mathtt{prop} \setminus \mathtt{id}) \cap \mathtt{int}) \cup \mathtt{ppo} \cup \mathtt{rfe} \\
\mathtt{pb} := \mathtt{prop} \,;\, \mathtt{strong\text{-}fence} \,;\, \mathtt{hb}^{*}
\end{array}
$$

Figure 8. LK definitions

3.2.1 "[I]f some [. . .] architecture gets its memory ordering wrong [. . .], [it] should pay the price" [80]

Some architectures do not provide sufficient ordering for the LK. The LK compensates in architecture-specific ways, and our LK model reflects only the ordering provided by the hardware. A notable example is Itanium, which can reorder loads from the same address. To work around this, Itanium's gcc compiler emits special load instructions, which provide suitable ordering guarantees for `READ_ONCE`. Accordingly, even though all other architectures' compilers need only emit a plain load, our LK model requires memory accesses to be annotated by once or something stronger (see Table 3).

3.2.2 The preserved program order relation `ppo`

`ppo` relates events in program order as described below:

Local orderings to writes are modeled by `to-w`. The `rwdep` (*read-write dependency*) relation orders a read and a write with an address, data, or control dependency between them (`dep`∪`ctrl`) (see [37, l. 879]). In Figure 4, there is a control dependency between a and b ((a, b) ∈ `ctrl`); thus (a, b) ∈ ppo.

The `overwrite` relation orders events where the second overwrites the first. Among the local orderings to writes, we consider only the instances of `overwrite` that are internal to a thread; hence the intersection with the `int` relation.

Local orderings to reads are modeled by the `to-r` relation. Read-read dependencies (formalised by `rrdep`) consist of `addr`, or `dep` followed by `rfi` (internal reads-from) (see [37, l. 393]). Unfortunately, Alpha does not respect read-read address dependencies [18]. The LK compensates via `smp_read_barrier_depends` (modeled by `rb-dep`), which emits a memory barrier on Alpha and is a no-op on other architectures. Our model therefore respects read-read dependencies only given an intervening `smp_read_barrier_depends` (see [37, l. 429, l. 550]), as modeled by `strong-rrdep` (*strong read-read dependency*).

An internal reads-from between a write release and a read acquire also provides ordering.

Local ordering due to fences is modeled by the `fence` relation (see [37, l. 1801]). The `strong-fence` relation orders events separated by `smp_mb`; we will update it in the next section to account for RCU. The `fence` relation orders events separated by a fence (`mb` [37, l. 446], `smp_wmb` [37, l. 1801] or `smp_rmb` [37, l. 1801]), or such that the first event is an `acquire` (acq-po) [37, l. 461] or the second is a `release` (po-`rel`) [37, l. 477]. In Figure 5, d and e are separated by an `smp_rmb` fence (i.e., (d, e) ∈ `rmb`); thus (d, e) ∈ `fence`. In Figure 7, d is a write release; thus (c, d) ∈ `po-rel` ⊆ `fence`.

ARMv7 implements `smp_load_acquire` with a full fence for lack of better means. In contrast, Power uses the lightweight `lwsync`, and ARMv8 a special load-acquire. Our model represents `smp_load_acquire` with the weaker orderings of ARMv8 [47] and Power [12], not the stronger ordering of ARMv7 [12]. The situation for `smp_store_release` is the same.

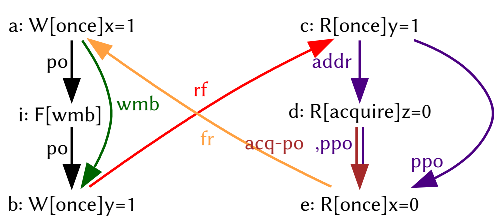

Figure 9. MP+wmb+addr-acq: Forbidden.

All in all, ppo pairs events linked by one of the relations above, optionally preceded by a read-read dependency (in the sense of `rrdep`). The LK uses this prefix (as documented in `task_rq_lock` [82, `kernel/sched/core.c`]) to forbid Figure 9: d is address-dependent (`addr` arrow) on c, thus (c, d) ∈ `rrdep`; and d is an acquire, thus (d, e) ∈ `acq-po`, which entails (d, e) ∈ `strong-rrdep` ⊆ `to-r`. Therefore (c, e) ∈ ppo.

3.2.3 The propagation relation prop

This relation corresponds to the informal notion of *transitivity* presented in [37, l. 1349]. It pairs events possibly in different threads ordered as follows.

Cumulative fences are modeled by `cumul-fence`, which pairs events in program order that are separated by an `smp_wmb` or `smp_mb` fence, or where the second is a release.

Strong fences (`smp_mb`) and releases are *A-cumulative*, as formalised by the cat function `A-cumul`(r) := `rfe`$^?$; r. The ordering provided by these fences extends to external writes that are read by an event preceding the fence. In Figure 5, c in T_1 is a write release, thus (b, c) ∈ `po-rel`. Since b reads the write a in T_0, (a, b) ∈ `rfe` and thus (a, c) ∈ `A-cumul(po-rel)`; hence (a, c) ∈ `cumul-fence`.

The relation* prop *generalises cumulativity: it ensures that guarantees made by `cumul-fence` for a thread T spread to other threads that access the same variables as T. When e_1 and e_2 are related by a sequence of `cumul-fence` links:

- (e_1, e_2) ∈ prop. In Figure 2, a and b are separated by an `smp_wmb` fence; thus they are related by prop.
- Any external event overwritten by e_1 links by prop to e_2. In Figure 2, d is overwritten by a; thus (d, b) ∈ prop.
- e_1 (or an external event it overwrites) is related by prop to events that read from e_2. In Figure 7, b is overwritten by c and the release d is read by e; thus (b, e) ∈ prop.
- These facts hold when $e_1 = e_2$. For instance, in Figure 6, d is overwritten by a; thus (d, a) ∈ prop. Idem f and a in Figure 7.

3.2.4 The happens-before relation hb

hb is the union of the ppo and `rfe` relations, together with prop restricted to distinct events in the same thread. The Hb axiom requires hb to be acyclic, ensuring that reads-from is consistent with local orderings due to ppo and fences.

In Figure 4, we have (a, b) ∈ `ctrl`; thus (a, b) ∈ ppo (as `ctrl` ⊆ `to-w` ⊆ ppo). We also have (c, d) ∈ mb; thus (c, d) ∈ ppo (as mb ⊆ `fence` ⊆ ppo). Overall, we have a $\xrightarrow{\text{ppo}}$ b $\xrightarrow{\text{rfe}}$ c $\xrightarrow{\text{ppo}}$ d $\xrightarrow{\text{rfe}}$ a, a cycle in the hb relation.

In Figure 5, we have (a, c) ∈ `cumul-fence`, as mentioned above. Moreover, a overwrites e and d reads from c; thus (e, d) ∈ prop. Since e and d are different events in the same thread, we have (e, d) ∈ (prop \ id) ∩ `int`. And since d and e are separated by an `rmb` fence, we also have (d, e) ∈ ppo. Thus d $\xrightarrow{\text{ppo}}$ e $\xrightarrow{\text{(prop\textbackslash id)}\cap\text{int}}$ d, a cycle in hb.

3.2.5 The propagates-before relation pb

pb contains events related by prop followed by a strong fence and an arbitrary number of hb links. The Pb axiom requires pb to be acyclic, so that events are overwritten in a manner consistent with the orderings due to strong fences.

In Figure 6, (d, a) ∈ prop, as mentioned above. Since a and b are separated by a strong fence, we have (d, b) ∈ pb. By symmetry we also have (b, d) ∈ pb, hence a cycle in pb.

In Figure 7, (b, e) ∈ prop, as mentioned above. Since e and f are separated by a strong fence, we have (b, f) ∈ pb. Similarly, since (f, a) ∈ prop and (a, b) ∈ strong-fence, we also have (f, b) ∈ pb, thus creating a cycle in pb.

3.3 Summary

This section presented the core of our formal LK model.

3.3.1 Our core LK model (Figure 3)

We exclude executions exhibiting any of following cycles:

- Scpv cycles, which involve only one shared variable, made of program order and communications edges;
- At cycles, which involve read-modify-writes;
- Hb cycles, which involve local orderings due to dependencies and fences, and reads-from communications;
- Pb cycles, which involve at least one strong fence.

3.3.2 The relations constrained by the model

These are formally defined in Figure 8. The crucial ones are:

- preserved program order ppo, which models local orderings due to dependencies (to-r and to-w) and fences;
- the propagation relation prop, which models the effect of fences (cumul-fence) on the propagation of writes to different variables with respect to one another;
- the happens-before relation hb, which models the effect of local orderings due to ppo and fences on reads-from;
- the propagates-before relation pb, modeling strong fences.

4 Modeling Read-Copy-Update

Table 4. RCU primitives and corresponding events

LK/C primitive	Event
rcu_dereference()	R[once], F[rb-dep]
rcu_assign_pointer()	W[release]
rcu_read_lock()	F[rcu-lock]
rcu_read_unlock()	F[rcu-unlock]
synchronize_rcu()	F[sync-rcu]

Read-copy update (RCU) is a synchronisation mechanism in which writers do not block readers: readers can be fast and scalable and writers can make forward progress concurrently with readers. Readers call the primitives rcu_read_lock and rcu_read_unlock to delimit a *read-side critical section* (RSCS). *Updaters* are writers that call the synchronize_rcu primitive; calling it starts a *grace period* (GP). Table 4 lists RCU primitives and their corresponding events.

In Figure 10, T_0 contains an RSCS accessing variables x and y, and T_1 updates the same variables.

We present here two different ways of formalising RCU: the fundamental law in Section 4.1 and the RCU axiom in

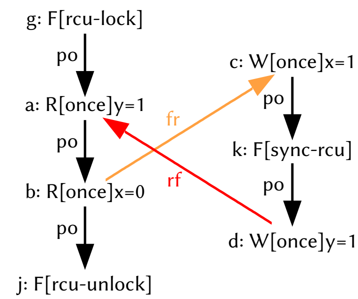

Figure 10. RCU-MP: Forbidden.

Section 4.2. We show the equivalence of the law and the axiom in Theorem 1. This result has practical significance because it enables tools to embed RCU semantics in either of two ways: by determining if a critical section spans a grace period (as per the law), or by counting the number of grace periods and critical sections in a cycle (as per the axiom).

4.1 Formalising the fundamental law of RCU

In [62], "an informal, high-level specification for RCU", the reader is warned thus:

RCU's specification is primarily empirical in nature

We would like to formalise the requirement of [62], i.e., the *fundamental law of RCU* (aka *grace period guarantee*) [66]:

Read-side critical sections cannot span grace periods.

Intuitively, for any GP and RSCS, the law has two aspects:

- RSCS precedes GP: if any access in the RSCS precedes the GP, then no access in the RSCS can follow the GP.
- GP precedes RSCS: if any access in the RSCS follows the GP, then no access in the RSCS can precede the GP.

We illustrate each aspect below, referring to Figure 10.

RSCS precedes GP: we take the fr arrow to indicate that b precedes c, hence b precedes the synchronize_rcu event k. Thus an access in the RSCS precedes the GP. By the fundamental law, no access in the RSCS can follow the GP. Thus a cannot read from d, which forbids the pattern.

GP precedes RSCS: we take the rf arrow to indicate that a follows d, i.e., a executes after synchronize_rcu returns. The law says that no access in the RSCS can precede the GP. Thus b cannot precede c, which again forbids the pattern.

The guarantees made by the law may seem similar to the ones made by fences. Indeed, the pattern of Figure 10 would also be forbidden with wmb in T_1 and rmb in T_0 (cf. Figure 1). However, unlike with fences, if we swap the reads in T_0 (cf. Figure 11) the pattern remains forbidden: if the read of x obtains 0 and hence executes before the GP, then the read of y cannot obtain 1.

We model the law with a "precedes" function F which, given a candidate execution, an RSCS, and a GP, selects which of the RSCS or the GP precedes the other:

$$F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{RSCS} \quad \text{or} \quad F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{GP}.$$

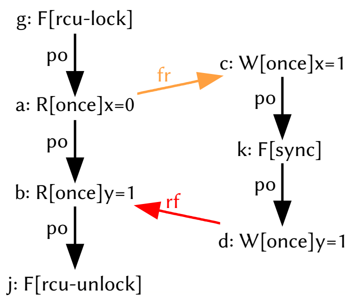

Figure 11. RCU-deferred-free: Forbidden.

The `rcu-fence`(F) relation models the interaction of an RSCS and a GP. Events, e_1 and e_2, are related by `rcu-fence`(F), if and only if there are an RSCS (delimited by `rcu_read_lock` and `rcu_read_unlock` events l and u) and a GP (given by `synchronize_rcu` event s) such that either:

- the RSCS precedes the GP, e_1 precedes u in program order, and e_2 is s itself or follows s in program order: $F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{RSCS} \wedge (e_1, u) \in \mathsf{po} \wedge (s, e_2) \in \mathsf{po}^?$
- or the GP precedes the RSCS, e_1 precedes s in program order, and e_2 is l itself or follows l in program order: $F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{GP} \wedge (e_1, s) \in \mathsf{po} \wedge (l, e_2) \in \mathsf{po}^?$

Let us revisit Figure 10 in the light of our new definition.

- If $F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{RSCS}$ (i.e., the RSCS precedes the GP), all events preceding the unlock event j in program order are related by `rcu-fence`(F) to the GP event k and all po-subsequent events. In particular, we have $(\mathrm{a}, \mathrm{d}) \in$ `rcu-fence`(F).
- If $F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{GP}$ (i.e., the GP precedes the RSCS), all events preceding the GP event k are related by `rcu-fence`(F) to the lock event g and all po-subsequent events. In particular, we have $(\mathrm{c}, \mathrm{b}) \in$ `rcu-fence`(F).

The fundamental law makes guarantees similar to fences, albeit stronger. Thus we treat `rcu-fence`(F) on a par with `strong-fence` and embed it in an enlarged pb(F) relation:

$$\mathsf{pb}(F) := \mathsf{prop}; (\texttt{strong-fence} \cup \texttt{rcu-fence}(F)); \mathsf{hb}^*$$

A candidate execution X *satisfies the fundamental law of RCU* iff there is a precedes function F such that X satisfies the enlarged Pb axiom acyclic(pb(F)). We see that there is no such function for the execution in Figure 10:

- if $F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{RSCS}$ then $(\mathrm{a}, \mathrm{d}) \in$ `rcu-fence`(F). Moreover we have $(\mathrm{d}, \mathrm{a}) \in$ `rfe`, thus in hb^*. This creates a cycle in pb(F).
- if $F(\mathrm{RSCS}, \mathrm{GP}) = \mathrm{GP}$ then $(\mathrm{c}, \mathrm{b}) \in$ `rcu-fence`(F). Moreover we have $(\mathrm{b}, \mathrm{c}) \in$ `fre`, thus in prop. This also creates a cycle in pb(F).

4.2 The RCU axiom

We augment our model with the relations in Figure 12.

We write gp for the relation between events in program order separated by a `synchronize_rcu` s, or such that the

```
gp := (po ∩ (_ × Sync)) ; po?
strong-fence := mb ∪ gp
rscs := po ; crit⁻¹ ; po?
link := hb* ; pb* ; prop
gp-link := gp ; link
rscs-link := rscs ; link
rec rcu-path := gp-link ∪ (rcu-path ; rcu-path)
    ∪ (gp-link ; rscs-link) ∪ (rscs-link ; gp-link)
    ∪ (gp-link ; rcu-path ; rscs-link)
    ∪ (rscs-link ; rcu-path ; gp-link)
irreflexive(rcu-path)                                (RCU)
```

Figure 12. RCU relations and axiom

second one is s itself. In Figure 10, we have (c, k) and (c, d) in gp. We add gp to the definition of `strong-fence`, so that `synchronize_rcu` can be used instead of `smp_mb`.

We write `crit` for the relation between an RSCS's lock l and its unlock u. The LK allows `rcu_read_lock()` and `rcu_read_unlock()` to be nested arbitrarily deeply; `crit` connects each outermost `rcu_read_lock()` to its matching `rcu_read_unlock()`. We omit its definition for brevity.

The relation `rscs` pairs events e_1 and e_2 in the same thread s.t. e_1 is po-before an unlock u and e_2 is po-after the matching lock l or is l itself. In Figure 10, (g, g), (g, a), (a, b), (b, a), $(\mathrm{b}, \mathrm{zj})$, (b, g), and many other pairs are in `rscs`.

The `link` ***relation*** embeds everything that provides order in our model. Intuitively, if an event in an RSCS appears before a GP according to our `link` relation, we model the first aspect of the fundamental law; if a GP appears before an event in an RSCS in `link`, we model the second aspect.

The `gp-link` ***and*** `rscs-link` ***relations*** are gp followed by `link` and `rscs` followed by `link`, respectively. Roughly speaking, they pair events where the second occurs after a GP following, or RSCS containing, the first.

The `rcu-path` ***relation*** is defined recursively, as indicated by the cat keyword `rec`. It merely pairs events that are connected by a non-empty sequence of `gp-link` and `rscs-link` edges in which there are at least as many GPs as RSCSes.

The **`RCU`** ***axiom*** requires `rcu-path` to be a path, i.e., to be irreflexive. Strikingly, our work allows us to demonstrate that this is equivalent to the fundamental law:

Theorem 1 (RCU guarantee)**.** *An LK candidate execution satisfies the* Pb *and* RCU *axioms iff it satisfies the fundamental law.*

This theorem formalises a rather simple rule of thumb [65, slide 42]: the fundamental law of RCU is invalidated iff there is a cycle in which the number of RSCSes is less than or equal to the number of GPs.

To establish this result, we show that the irreflexivity of `rcu-path` (as per the axiom) is equivalent to the acyclicity of pb enlarged by `rcu-fence`(F) (as per the law). We omit the proof (available online [7]) for brevity.

	Model	Power8	ARMv8	ARMv7	X86	C11
LB	Allow	0/15G	0/10G	0/3.0G	0/33G	Allow
LB+ctrl+mb, Fig. 4	Forbid	0/17G	0/5.1G	0/4.4G	0/18G	Allow
WRC	Allow	741k/7.7G	13k/5.2G	0/1.6G	0/17G	Allow
WRC+wmb+acq, Fig. 14	Allow	0/7.5G	0/4.6G	0/1.6G	0/16G	Forbid
WRC+po-rel+rmb, Fig. 5	Forbid	0/7.7G	0/5.3G	0/1.6G	0/17G	Forbid
SB	Allow	4.4G/15G	2.4G/10G	429M/3.0G	765M/33G	Allow
SB+mbs, Fig. 6	Forbid	0/15G	0/10G	0/3.0G	0/33G	Forbid
MP	Allow	57M/15G	104M/10G	3.0M/3.0G	0/33G	Allow
MP+wmb+rmb, Fig. 2	Forbid	0/15G	0/9.4G	0/2.6G	0/33G	Forbid
PeterZ-No-Synchro	Allow	26M/4.6G	3.6M/900M	10k/380M	351k/7.2G	Allow
PeterZ, Fig. 7	Forbid	0/8.7G	0/2.5G	0/2.2G	0/9.1G	Allow
RCU-deferred-free, Fig. 11	Forbid	0/256M	0/576M	0/15M	0/25M	—
RCU-MP, Fig. 10	Forbid	0/672M	0/336M	0/336M	0/336M	—
RWC	Allow	88M/11G	94M/4.8G	7.5M/1.6G	5.6M/17G	Allow
RWC+mbs, Fig. 13	Forbid	0/8.7G	0/2.5G	0/2.2G	0/9.1G	Allow

Table 5. Simulations vs. experimental results.

5 Experiments

We used the diy+herd toolsuite [5] to build a vast library of *litmus tests* and run them against our model and as kernel modules. We also compared our model to the C11 model of [15].

Litmus tests are small programs that exercise specific features of consistency models. Our validation includes classic tests [12, 13, 34, 38, 74, 75], new hand-written tests, and systematic variations of several tests (see e.g. [50, Sect. 9.1]) with all combinations of fences or dependencies. We used the diy7 tool [5] to systematically generate thousands of tests with cycles of edges (e.g., dependencies, reads-from, coherence) of increasing size. The tests, written in a subset of C supplemented with LK constructs such as `READ_ONCE` or `WRITE_ONCE`, are online [7].

Running litmus tests against cat models was carried out with the herd7.43 tool [5]. The herd tool can simulate any cat model, but initially supported only machine-level models of CPUs and GPUs [2, 6, 12] and language-level models for C11 and OpenCL [15]. We extended herd with support for the LK constructs used in our tests.

Running litmus tests as kernel modules was done using our new klitmus tool, inspired by the litmus tool [5]. The new tool differs from litmus in that kernel programming is different from userspace programming: we had to find LK equivalents to the userspace libraries used by litmus. E.g., launching threads is performed using LK `kthreads` instead of userspace `pthreads`. The test results cannot be sent to standard output, so we instead read the kernel module's output via the `/proc` filesystem.

5.1 Hardware results

We tested a CHRP IBM pSeries with 8 POWER8E CPUs at 3.4GHz (Linux v4.4.40), an Amlogic ARMv8 with 4 Cortex-A53 cores at 1.5GHz (Linux v3.14.29), a Raspberry Pi ARMv7 with 4 Cortex-A7 cores at 900Mhz (Linux v4.9.20), and an HP desktop with 2 (6-core) Intel Xeon E5-2620 v3 CPUs at 2.40GHz (Linux v3.16.04).

Table 5 summarises our results; the complete set is at [7]. For each test we give the number of times it was observed on hardware, over the times it was run: k stands for 10^3, M for 10^6 and G for 10^9. E.g., we ran LB+ctrl+mb (Figure 4) 17G times on Power8, but never observed it. This is expected, as the model forbids the idiom.

Indeed, a result observed during experiments but forbidden by the model indicates a bug. One cannot make definite conclusions from the absence of observation, but the tool proved rather discriminating [2, 10–12, 74, 75].

For reference, we include tests without synchronisation. E.g., Figure 4 shows LB+ctrl+mb with a control dependency and an `mb` fence; its sibling LB has no dependency and no fence.

Table 5 shows that all the hardware behaviours we observed are allowed by the model: our model is experimentally sound. Some behaviours allowed by the model have not been observed in experiments; the machines are stronger than required by our model. For instance, LB, although allowed by our model, was not observed on any of our systems. It was observed on other ARMv7 machines, however [50, Sect. 7.1].

5.2 Comparison to C11

To compare our LK model and C11, we used the cat model of [15], and the mapping from LK primitives to C11 primitives given in [68]. The complete results are available at [7]. Our experiments quantify the differences between LK (see first column of Table 5) and C11 (see last column), using this mapping.

For example, `smp_mb` "restores SC", but its C11 counterpart `atomic_thread_fence(memory_order_seq_cst)` does not.

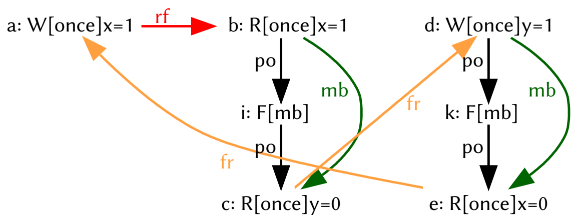

Figure 13. RWC+mbs: Forbidden.

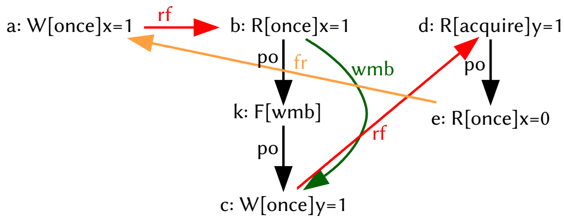

Figure 14. WRC+wmb+acq: Allowed.

As an example of this difference, the LK model forbids the pattern in Figure 13 (there is a cycle in pb) but C11 allows it.

In fact, no known production-quality implementation of C11 fails to forbid Figure 13 [43, 74]. But originally, C11 allowed it so that the `seq_cst` fence could be implemented with Itanium's `mf` instruction. Eventually `relaxed` loads were defined to forbid reordering of loads to the same variable, forcing Itanium to generate `ld,acq` for `relaxed` loads; hence `mf` is now sufficient to forbid Figure 13. The current consensus is that C11's fence should be strengthened to restore SC (as `smp_mb` does in the LK); there are various ideas on how to accomplish this [15, 44].

There are other differences: the LK respects control dependencies between a read and a write (ctrl ⊆ to-w ⊆ ppo in Figure 8), thus forbidding the outcome of Figure 4, which C11 allows. Moreover, the test WRC+wmb+acq (Figure 14), which C11 forbids but the LK allows, shows that there is no ideal equivalent of `smp_wmb` in C11 [68].

6 Verifying an RCU implementation

The RCU implementation in Figure 15, used in the Linux trace tool [1], provides code for `rcu_read_lock` (lines 8 to 18), `rcu_read_unlock` (lines 20 to 25) and `synchronize_rcu` (lines 43 to 50). We explain here why it satisfies the fundamental law of RCU at a high level, and refer the reader to [7] for details.

6.1 Description of the implementation

Threads communicate via an array of variables `rc[]` (line 4) and a grace-period control variable `gc` (line 5). The `gp_lock` mutex (line 6) serialises grace periods. The `GP_PHASE` (line 1) bit of `gc` indicates which *phase* a grace period is in (grace periods have two phases). The low-order bits of `rc[i]` selected by `CS_MASK` (line 2) form a 16-bit *counter*.

***The counter in* `rc[i]`** records the depth of RSCS nesting for thread `i`: initially 0, set to 1 at line 13 in an outermost `rcu_read_lock` call, incremented at line 16 in inner calls, and decremented at line 24 in `rcu_read_unlock`. If RSCSes are properly nested (no unlock without a earlier matching lock) and the depth of nesting does not overflow the 16-bit counter, only an outermost `rcu_read_unlock` sets the counter to 0, indicating that thread `i` is not in an RSCS.

***The* `GP_PHASE` *bit in* `gc`** is 0 before a grace period, viz, before `synchronize_rcu` is called. That routine sets the phase to 1 and then 0 again (line 36). Threads starting an RSCS copy the current phase value into their respective `rc[i]` (line 13). Thus `synchronize_rcu` knows which threads must be waited for. Indeed, after changing the phase, `update_counter_and_wait` waits for each thread `i` (lines 38–39) until the value computed at lines 29–30 becomes false. This happens when:

- `rc[i]`'s counter is zero (thread `i` is not in an RSCS), or
- `rc[i]`'s counter is nonzero and its phase bit is equal to that of `gc` (thread `i` is in an RSCS which started after the current GP phase).

6.2 Correctness statement

Let P be an LK program, and let P' be obtained by replacing the RCU primitives in P with the routines of Figure 15. For any execution X' of P' allowed by our model, let X be the corresponding execution of P. Each non-RCU event e in X corresponds directly to an event e' in X'. (Consider, e.g., the execution X in Figure 10, corresponding to X' in Figure 16. Events a, b, c, and d in X match `a'`, `b'`, `c'`, and `d'` in X'.)

Furthermore, since the code in Figure 15 does not access any of the shared locations in P, and conversely, P does not access the shared locations `gc` and `rc[]`, each read in X is related by `rf` in X' to a write also in X. (For example, `a'` in X' reads from `d'`, not from some other write present in X' but not in X.) More generally, the non-RCU relations of X are simply those of X' restricted to the events in X.

We set up a similar correspondence for the RCU events (in Figure 16, appended to these events' labels are the line numbers from Figure 15 for the events and their call chains):

- For each `F[rcu_lock]` event l in X (g in Figure 10), let l' be the write of `rc[i]` at line 13 (or 16 for inner nesting levels). In Figure 16, this is `g'`.
- For each `F[rcu_unlock]` event u in X (`j` in Figure 10), let u' be the write of `rc[i]` at line 24 (`j'` in Figure 16).
- For each `F[sync-rcu]` event s in X (`k` in Figure 10), let s' be the write to `gc` at line 36, from the call to `update_counter_and_wait` at line 46 (`k'` in Figure 16).

We can now state our correctness result:

Theorem 2 (Correctness of RCU implementation)**.** *If X' is allowed in our LK model and has properly nested RSCSes that do not overflow the counters in* `rc[]`*, then X is allowed.*

```
1  #define GP_PHASE 0x10000
2  #define CS_MASK  0x0ffff
3
4  static unsigned long rc[MAX_THREADS] = {0};
5  static unsigned long gc = 1;
6  static DEFINE_MUTEX(gp_lock);
7
8  void rcu_read_lock(void) {
9     unsigned int i = get_my_tid();
10    unsigned long tmp = READ_ONCE(rc[i]);
11
12    if (!(tmp & CS_MASK)) {
13       WRITE_ONCE(rc[i], READ_ONCE(gc));
14       smp_mb();
15    } else {
16       WRITE_ONCE(rc[i], tmp + 1);
17    }
18 }
19
20 void rcu_read_unlock(void) {
21    unsigned int i = get_my_tid();
22
23    smp_mb();
24    WRITE_ONCE(rc[i], READ_ONCE(rc[i]) - 1);
25 }
26 static int gp_ongoing(unsigned int i) {
27    unsigned long val = READ_ONCE(rc[i]);
28
29    return (val & CS_MASK)
30       && ((val ^ READ_ONCE(gc)) & GP_PHASE);
31 }
32
33 static void update_counter_and_wait(void) {
34    unsigned int i;
35
36    WRITE_ONCE(gc, READ_ONCE(gc) ^ GP_PHASE);
37    for (i = 0; i < MAX_THREADS; i++) {
38       while (gp_ongoing(i))
39          msleep(10);
40    }
41 }
42
43 void synchronize_rcu(void) {
44    smp_mb();
45    mutex_lock(&gp_lock);
46    update_counter_and_wait();
47    update_counter_and_wait();
48    mutex_unlock(&gp_lock);
49    smp_mb();
50 }
```

Figure 15. RCU implementation from [29].

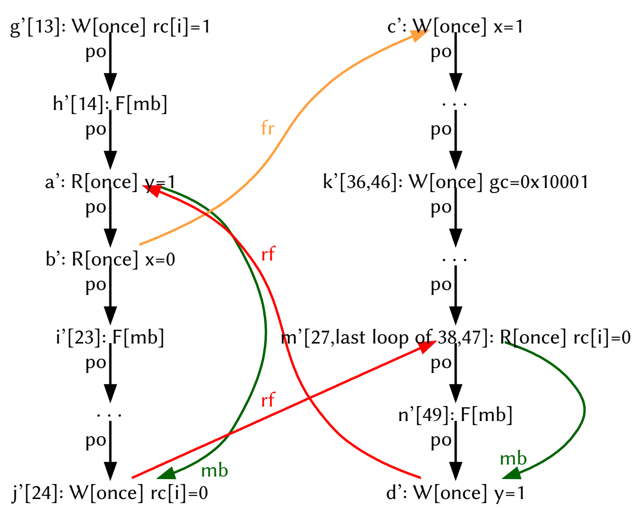

Figure 16. RCU-MP, with RCU as implemented in Figure 15

6.3 Proof sketch

For brevity, we only list critical points of our proof; detailed proofs are in [7].

***All non-RCU relations** R **in** X **hold in** X'**:** when $(e_1, e_2) \in R$ holds in X, the corresponding fact $(e'_1, e'_2) \in R$ holds in X'. Recall that we defined X to differ from X' only for RCU events and relations. Hence this result is immediate except when R is `strong-fence`, which contains the RCU relation gp. Fortunately it is true in this case as well.

To see why, consider $(e_1, e_2) \in$ gp in X (e.g., the writes c and d in Figure 10). There is an `F[sync-rcu]` event between them in program order; hence the `F[mb]` event arising from line 44 lies between the corresponding events e'_1 and e'_2 in X'. Thus $(e'_1, e'_2) \in$ mb, implying that $(e'_1, e'_2) \in$ `strong-fence`. (Between `c'` and `d'` in Figure 16 are all the events from Figure 15's implementation of `synchronize_rcu`; the `F[mb]` event for line 44 is elided.)

Since X' is allowed, X thus obeys all the core constraints of our model, leaving only the RCU constraint to consider.

Using our RCU guarantee theorem (Section 4.2), we show that X does obey the RCU constraint by showing that X satisfies the fundamental law of RCU. This requires finding a precedes function F for X such that pb(F) is acyclic.

Our precedes function is derived from the execution X'. Given a GP in X and an outermost RSCS in thread i, let l and u be the lock and unlock of the RSCS. The corresponding events l' and u' in X' were defined in Section 6.2. We consider two distinguished read events, r_1 and r_2, where:

- r_1 is the read of `rc[i]` executed by line 27 of Figure 15,
- in the call to `gp_ongoing(i)` from the last iteration of the `while` loop at line 38,
- in the first call to `update_counter_and_wait` (line 46) within the GP,

and r_2 is the equivalent read from within the second call to `update_counter_and_wait` (line 47). In Figure 16, r_2 = `m'` and r_1 is not shown.

At least one of the following two facts must hold in X':

1. the RSCS's `rcu_read_lock` was not visible at the start of the GP: $(r_1, l') \in \texttt{fr}$;
2. the RSCS's `rcu_read_unlock` or a later write to `rc[i]` was visible at the end of the GP: $(u', r_2) \in (\texttt{coi}^? ; \texttt{rf})$.

We take $F(\text{RSCS}, \text{GP})$ to be GP if (1) holds and RSCS otherwise. In Figure 16, (2) holds since u' is `j'`, r_2 is `m'`, and $(\texttt{j'}, \texttt{m'}) \in \texttt{rf}$. Thus $F(\text{RSCS}, \text{GP}) = \text{RSCS}$.

***A cycle in* $\texttt{pb}(F)$ *for* X** would give rise to a cycle in `pb` for X'. We omit the full demonstration (given in [7]) but illustrate it with our example. We know from Section 4.1 that X in Figure 10 violates the fundamental law of RCU and every $\texttt{pb}(F)$ relation for X contains a cycle. We are now claiming this means that X' in Figure 16 has a cycle in `pb`. And so it does: $\texttt{d'} \xrightarrow{\texttt{rfe}} \texttt{a'} \xrightarrow{\texttt{mb}} \texttt{j'}$, hence $\texttt{d'} \xrightarrow{\texttt{pb}} \texttt{j'}$, and similarly, $\texttt{j'} \xrightarrow{\texttt{pb}} \texttt{d'}$ via `m'`.

Returning to the general proof of Theorem 2: The theorem assumed that X' is allowed in our model and hence obeys the `Pb` constraint. This requires the `pb` relation in X' to be acyclic, from which we now deduce that the $\texttt{pb}(F)$ relation in X must also be acyclic. By our earlier remark, this suffices to conclude the proof sketch.

7 Discussion

The process that led to our LK model was iterative, and both social and technical. We reviewed [37] and wrote an initial cat file formalising our understanding. We used the litmus tests of [37, 66, 67] to refine this model, and asked questions to hardware designers and LK maintainers [8, 9, 25, 65]. Later we modified the tools of the diy+herd toolsuite [5] to generate more tests, and run them as kernel modules. We referred to published models when available, e.g., ARMv8 [47], and architectural definitions of LK primitives [37, 69].

The need to account for all the architectures that the LK targets can make the model seem complex and arbitrary. For example, `smp_read_barrier_depends` exists exclusively for the sake of Alpha. Otherwise, in the definition of `ppo`, the relations `strong-rrdep` and `rrdep` would be the same.

We do think that our LK model is, perhaps surprisingly, less subtle than C11 and OpenCL [15], as it is inspired by hardware: thus our model does not have out-of-thin-air values, because it respects dependencies as hardware does; and the LK's full fence restores SC, unlike that of C11.

All in all, the model is as complex and arbitrary as the LK is. Consequently it is as stable as the LK is; we expect it to change as often as [37] does, i.e., a handful of times per year. The LK model will adapt as architectures change (or become better defined), as workloads change, and as kernel developers become more aggressive in their pursuit of performance and scalability.

To support existing non-buggy LK code, an LK model must account for the LK's primitives, including fences, RCU, read-modify-writes, and locking. Our work models all these primitives, except for locks. This is due to the current lack of consensus on the semantics of certain locking primitives [83] within the LK community, which our preliminary work on the topic helped uncover.

Locking may, however, be emulated with the constructs that we already have [63]. For example, we model a spinlock as a shared location. The `spin_lock` primitive behaves like `xchg_acquire` for this location. In Table 3, this is modeled as a read with annotation `acquire` and a write with annotation `once`, governed by the `At` axiom of Figure 3 and the constraints on `acquire` in Figure 8. The `spin_unlock` primitive behaves like a `smp_store_release` for the shared location, governed by the constraints on `release` in Figure 8.

Other features not currently supported by our model are:

- compiler optimizations (however, the LK's `READ_ONCE` and `WRITE_ONCE` rule out many optimizations [20, 21], so this limitation is less of a problem than it might seem);
- any kind of arithmetic;
- multiple access sizes and partially overlapping accesses;
- non-trivial data, including arrays and structures;
- dynamic memory allocation;
- exceptions, interrupts, self-modifying code, and I/O;
- asynchronous RCU grace period primitives, including `call_rcu` and `rcu_barrier`.

We do hope to address these limitations over time. But even in its current form, our model provides a reference for making decisions about concurrency in the LK, as witnessed by the issues that our work helped discuss or settle (Table 2).

Acknowledgements We thank H. Peter Anvin, Will Deacon, Andy Glew, Derek Williams, Leonid Yegoshin, and Peter Zijlstra for their patient explanations of their respective systems' models; Boqun Feng and Mark Rutland for their keen interest and suggestions; Sylvan Clebsch, Will Deacon and Daryl Stewart for comments on a draft; and Jessica Murillo and Mark Figley for their support of this effort. Finally, we thank our reviewers, especially our shepherd Dan Lustig, for their helpful and enthusiastic reviews.

References

[1] 2017. Linux Trace Tool (LTTng). http://lttng.org/. (2017).
[2] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *ASPLOS 2015*.
[3] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language cat. *CoRR* abs/1608.07531 (2016). http://arxiv.org/abs/1608.07531
[4] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software.

In *Computer Aided Verification (CAV) (LNCS)*, Vol. 8044. Springer, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

[5] Jade Alglave and Luc Maranget. 2011–2017. The diy7 tool suite. http://diy.inria.fr/. (2011–2017).

[6] Jade Alglave and Luc Maranget. 2015. Towards a formalisation of the HSA memory model in the cat language. http://www.hsafoundation.com/?ddownload=5381. (2015).

[7] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2017. A formal model of Linux-kernel memory ordering – companion webpage. http://diy.inria.fr/linux/. (2017). [Online; accessed 25-December-2017].

[8] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2017. A formal kernel memory-ordering model (part 1). (14 April 2017). https://lwn.net/Articles/718628/.

[9] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2017. A formal kernel memory-ordering model (part 2). (20 April 2017). https://lwn.net/Articles/720550/.

[10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 258–272.

[11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in weak memory models (extended version). *Formal Methods in System Design* 40, 2 (2012), 170–205.

[12] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 7:1–7:74 pages.

[13] ARM. 2009. *ARM Barrier Litmus Tests and Cookbook*. ARM Ltd.

[14] ARM. 2014. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd.

[15] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 634–648.

[16] Anton Blanchard. 2011. RE: [PATCH] smp_call_function_many SMP race. https://lkml.org/lkml/2011/1/11/489. (12 January 2011).

[17] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

[18] Compaq. 2002. *Alpha Architecture Reference Manual*. Compaq Computer Corporation.

[19] Jonathan Corbet. 2008. The lockless page cache. https://lwn.net/Articles/291826/. (29 July 2008).

[20] Jonathan Corbet. 2012. ACCESS_ONCE(). https://lwn.net/Articles/508991/. (1 August 2012).

[21] Jonathan Corbet. 2014. ACCESS_ONCE() and compiler bugs. https://lwn.net/Articles/624126/. (3 December 2014).

[22] Jonathan Corbet. 2014. C11 atomic variables and the kernel. https://lwn.net/Articles/586838/. (18 February 2014).

[23] Jonathan Corbet. 2014. C11 atomics part 2: "consume" semantics. https://lwn.net/Articles/588300/. (26 February 2014).

[24] Jonathan Corbet. 2016. Time to move to C11 atomics? https://lwn.net/Articles/691128/. (15 June 2016).

[25] Michael Cree. 2017. Re: Question about DEC Alpha memory ordering. lkml.kernel.org/r/20170214192646.m6ydg27nwnh7bg7o@tower. (2017).

[26] Will Deacon. 2015. [PATCH] arm64: spinlock: serialise spin_unlock_wait against concurrent lockers. https://marc.info/?l=linux-arm-kernel&m=144862480822027. (2015).

[27] Will Deacon. 2015. Re: [PATCH] arm64: spinlock: serialise spin_unlock_wait against concurrent lockers. https://marc.info/?l=linux-arm-kernel&m=144898777124295. (2015).

[28] Will Deacon. 2016. [PATCH v2 1/3] arm64: spinlock: order spin_{is_locked, unlock_wait} against local locks. http://lists.infradead.org/pipermail/linux-arm-kernel/2016-June/434765.html. (2016).

[29] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Trans. Parallel Distrib. Syst.* 23, 2 (Feb. 2012), 375–382.

[30] Michael Ellerman. 2016. [PATCH v3] powerpc: spinlock: Fix spin_unlock_wait(). https://marc.info/?l=linux-kernel&m=146521336230748&w=2. (2016).

[31] Boqun Feng. 2015. Re: [PATCH 4/4] locking: Introduce smp_cond_acquire(). https://marc.info/?l=linux-kernel&m=144723482232285. (2015).

[32] Boqun Feng. 2016. [PATCH v2] powerpc: spinlock: Fix spin_unlock_wait(). https://marc.info/?l=linux-kernel&m=146492558531292&w=2. (2016).

[33] Boqun Feng. 2016. [PATCH v4] powerpc: spinlock: Fix spin_unlock_wait(). https://marc.info/?l=linuxppc-embedded&m=146553051027169&w=2. (2016).

[34] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 608–621.

[35] Mel Gorman. 2013. LWN Quotes of the week, 2013-12-11. http://lwn.net/Articles/575835/. (2013).

[36] Tejun Heo. 2010. [PATCH 3/4] scheduler: replace migration_thread with cpuhog. https://marc.info/?l=linux-kernel&m=126806371630498. (8 March 2010).

[37] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. 2017. Linux kernel memory barriers. https://www.kernel.org/doc/Documentation/memory-barriers.txt. (2017).

[38] IBM. 2009. *Power ISA Version 2.06*. IBM Corporation.

[39] Imagination. 2015. *MIPS®Architecture For Programmers, Volume II-A: The MIPS64®Instruction, Set Reference Manual*. Imagination Technologies, LTD. https://imgtec.com/?do-download=4302.

[40] Intel. 2002. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. Intel Corporation.

[41] Andi Kleen. 2013. Re: [PATCH v6 4/5] MCS Lock: Barrier corrections. https://marc.info/?l=linux-mm&m=138619237606428. (2013).

[42] Michalis Kokologiannakis and Konstantinos Sagonas. 2017. *Stateless Model Checking of the Linux Kernel's Hierarchical Read-Copy Update (Tree RCU)*. Technical Report. National Technical University of Athens. https://github.com/michalis-/rcu/blob/master/rcupaper.pdf.

[43] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 649–662.

[44] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[45] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

[46] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. 2016. Verification of the Tree-Based Hierarchical Read-Copy Update in the Linux Kernel. *CoRR* abs/1610.03052 (2016). http://arxiv.org/abs/1610.03052

[47] ARM Ltd. (Ed.). 2017. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Limited.

[48] Tony Luck. 2013. RE: Does Itanium permit speculative stores? https://marc.info/?l=linux-kernel&m=138427925823852. (12 November 2013).

[49] Tony Luck. 2013. RE: Does Itanium permit speculative stores? https://marc.info/?l=linux-kernel&m=138428203211477. (12 November 2013).

[50] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf. (Oct. 2012). Draft.

[51] Paul McKenney. 2001. RFC: patch to allow lock-free traversal of lists with insertion. https://lists.gt.net/linux/kernel/223665#223508. (2001).

[52] Paul McKenney. 2007. What is RCU, fundamentally? https://lwn.net/Articles/262464/. (2007).

[53] Paul McKenney. 2013. Re: [PATCH v6 4/5] MCS Lock: Barrier corrections. https://marc.info/?l=linux-mm&m=138540258209368. (2013).

[54] Paul McKenney. 2016. Re: [RFC][PATCH] mips: Fix arch_spin_unlock(). http://lkml.kernel.org/r/20160202120252.GI6719@linux.vnet.ibm.com. (2016).

[55] Paul McKenney. 2017. PROPER CARE AND FEEDING OF RETURN VALUES FROM rcu_dereference(). https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt. (2017).

[56] Paul E. McKenney. 2007. QRCU with lockless fastpath. https://lwn.net/Articles/223752/. (24 February 2007).

[57] Paul E. McKenney. 2013. Does Itanium permit speculative stores? https://marc.info/?l=linux-kernel&m=138419150923282. (11 November 2013).

[58] Paul E. McKenney. 2016. documentation: Present updated RCU guarantee. https://patchwork.kernel.org/patch/9428001/. (2016).

[59] Paul E. McKenney. 2016. documentation: Transitivity is not cumulativity. http://www.spinics.net/lists/linux-tip-commits/msg32905.html. (2016).

[60] Paul E. McKenney. 2016. Prototype patch for Linux-kernel memory model. http://lkml.kernel.org/r/20171113184031.GA26302@linux.vnet.ibm.com. (2016).

[61] Paul E. McKenney. 2016. [v3,11/41] mips: reuse asm-generic/barrier.h. https://patchwork.kernel.org/patch/8036201/. (14 January 2016).

[62] Paul E. McKenney. 2017. A tour through RCU's requirements. https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/Requirements.html. (2017).

[63] Paul E. McKenney. 2017. Re: [PATCH RFC 01/26] netfilter: Replace spin_unlock_wait() with lock/unlock pair. https://lkml.org/lkml/2017/6/27/1052. (2017).

[64] Paul E. McKenney. 2017. srcu: Force full grace-period ordering. https://patchwork.kernel.org/patch/9535987/. (2017).

[65] Paul E. McKenney, Jade Alglave, Luc Maranget, Andrea Parri, and Alan Stern. 2016. Linux-Kernel Memory Ordering: Help Arrives At Last!. In *LinuxCon Europe*. http://www.rdrop.com/users/paulmck/scalability/paper/LinuxMM.2016.10.04c.LCE.pdf.

[66] Paul E. McKenney, Mathieu Desnoyers, Lai Jiangshan, and Josh Triplett. 2013. The RCU-barrier menagerie. https://lwn.net/Articles/573497/. (2013).

[67] Paul E. McKenney, Mathieu Desnoyers, Lai Jiangshan, and Josh Triplett. 2013. User-space RCU: Memory-barrier menagerie. https://lwn.net/Articles/573436/. (2013).

[68] Paul E. McKenney, Ulrich Weigand, Andrea Parri, and Boqun Feng. 2016. Linux-Kernel Memory Model. (6 June 2016). http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0124r2.html.

[69] David S. Miller. 2017. Semantics and behavior of atomic and bitmask operations. https://www.kernel.org/doc/core-api/atomic_ops.rst. (2017).

[70] Milton Miller. 2011. [PATCH 0/4 v3] smp_call_function_many issues from review. https://marc.info/?l=linux-kernel&m=130021726530804. (15 March 2011).

[71] Ingo Molnar. 2013. Re: [PATCH v6 4/5] MCS Lock: Barrier corrections. https://marc.info/?l=linux-mm&m=138513336717990&w=2. (2013).

[72] Ingo Molnar. 2017. Re: [PATCH v2 0/9] Remove spin_unlock_wait(). https://marc.info/?l=linux-kernel&m=149942365927828&w=2. (2017).

[73] Jiri Olsa. 2009. [PATCHv5 2/2] memory barrier: adding smp_mb__after_lock. https://marc.info/?l=linux-netdev&m=124839648220382&w=2. (3 July 2009).

[74] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 311–322.

[75] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 175–186.

[76] Manfred Spraul. 2001. Re: RFC: patch to allow lock-free traversal of lists with insertion. http://lkml.iu.edu/hypermail/linux/kernel/0110.1/0410.html. (2001).

[77] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying Read-Copy-Update in a Logic for Weak Memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 110–120.

[78] Linus Torvalds. 2012. Re: Memory corruption due to word sharing. https://gcc.gnu.org/ml/gcc/2012-02/msg00013.html. (2012).

[79] Linus Torvalds. 2015. Re: [PATCH 4/4] locking: Introduce smp_cond_acquire(). http://lkml.kernel.org/r/CA+55aFyXu5iFJfdm7o-RKUm_9a850iSzeM+whmtUAotkY0EvTw@mail.gmail.com. (2015).

[80] Linus Torvalds. 2016. Re: [RFC][PATCH] mips: Fix arch_spin_unlock(). https://lkml.org/lkml/2016/2/2/80. (2016).

[81] Linus Torvalds. 2016. Re: [v3,11/41] mips: reuse asm-generic/barrier.h. https://marc.info/?l=linux-kernel&m=145384764324700&w=2. (26 January 2016).

[82] Linus Torvalds. 2017. Linux Kernel v4.12 (Fearless Coyote). https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.12.tar.xz. (2 July 2017).

[83] Linus Torvalds. 2017. Re: [GIT PULL rcu/next] RCU commits for 4.13. https://lkml.org/lkml/2017/6/27/1052. (2017).

[84] Srivatsa Vaddagiri. 2005. [PATCH] Fix RCU race in access of nohz_cpu_mask. http://lkml.iu.edu/hypermail/linux/kernel/0512.0/0976.html. (5 December 2005).

[85] Leonid Yegoshin. 2016. Re: [v3,11/41] mips: reuse asm-generic/barrier.h. https://marc.info/?l=linux-kernel&m=145263153305591&w=2. (12 January 2016).

[86] Leonid Yegoshin. 2016. Re: [v3,11/41] mips: reuse asm-generic/barrier.h. https://marc.info/?l=linux-kernel&m=145280444229608&w=2. (14 January 2016).

[87] Leonid Yegoshin. 2016. Re: [v3,11/41] mips: reuse asm-generic/barrier.h. https://marc.info/?l=linux-kernel&m=145280241129008&w=2. (14 January 2016).

[88] Peter Zijlstra. 2013. Re: Does Itanium permit speculative stores? https://marc.info/?l=linux-kernel&m=138428080207125. (12 November 2013).

[89] Peter Zijlstra. 2013. Re: [PATCH v6 4/5] MCS Lock: Barrier corrections. https://marc.info/?l=linux-mm&m=138514629508662&w=2. (2013).

[90] Peter Zijlstra. 2016. [tip:perf/urgent] perf/core: Fix sys_perf_event_open() vs. hotplug. https://www.spinics.net/lists/kernel/msg2421883.html. (14 January 2016).