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.

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

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]:


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 to 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 humanity 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 tool-suite [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

<table>
<thead>
<tr>
<th>LK issue</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>locking on ARM64</td>
<td>[26]</td>
</tr>
<tr>
<td>ambiguities in [37]</td>
<td>[59]</td>
</tr>
<tr>
<td>ambiguities in RCU documentation</td>
<td>[58]</td>
</tr>
<tr>
<td>CPU hotplug</td>
<td>[90]</td>
</tr>
<tr>
<td>assumption about lock-unlock</td>
<td>[64]</td>
</tr>
<tr>
<td>semantics of spin_unlock_wait</td>
<td>[83]</td>
</tr>
</tbody>
</table>

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. T0 updates x, calls _smp_wmb_, and sets y to 1. T1 reads y, calls _smp_rmb_, and reads x. This is a message passing idiom: with enough synchronisation, after T1 sees that the flag y is set, it must see the updated data. Here _smp_wmb_ and _smp_rmb_ are enough.

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**

<table>
<thead>
<tr>
<th>LK/C primitive</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_ONCE()</td>
<td>R[once]</td>
</tr>
<tr>
<td>WRITE_ONCE()</td>
<td>W[once]</td>
</tr>
<tr>
<td>_smp_load_acquire()</td>
<td>R[acquire]</td>
</tr>
<tr>
<td>_smp_store_release()</td>
<td>W[release]</td>
</tr>
<tr>
<td>_smp_rmb()</td>
<td>F[rmb]</td>
</tr>
<tr>
<td>_smp_wmb()</td>
<td>F[wmb]</td>
</tr>
<tr>
<td>_smp_mb()</td>
<td>F[mb]</td>
</tr>
<tr>
<td>_smp_read_barrier_depends()</td>
<td>F[rb-dep]</td>
</tr>
<tr>
<td>_xchg_relaxed()</td>
<td>R[once], W[once]</td>
</tr>
<tr>
<td>_xchg_acquire()</td>
<td>R[acquire], W[once]</td>
</tr>
<tr>
<td>_xchg_release()</td>
<td>R[once], W[release]</td>
</tr>
<tr>
<td>_xchg()</td>
<td>F[mb], R[once], W[once], F[mb]</td>
</tr>
</tbody>
</table>

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

- ONCE primitives prevent tearing (compiling a large access as a group of smaller accesses), 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
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, smpl_load_acquire is represented by a read annotated acquire, WRITE_ONCE by a write annotated once, and smpl_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 (\( \mathbb{W} \) contains all write events, \( \mathbb{R} \) all reads, and \( \mathbb{L} \) all events) and the relations forming candidate executions \( \mathbb{E} \) (\( po \), \( addr \), \( data \), \( ctrl \), \( rmw \), \( rf \), 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 (\( \neg \)), inverse (\( r^{-1} \)), reflexive closure (\( r^r \)), transitive closure (\( r^t \)), reflexive transitive closure (\( r^* \)), sequence (\( r_1 \circ r_2 \)), defined as \( \{ (x, y) \mid \exists z ((x, y) \in r_1 \land (y, z) \in r_2) \} \)), 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: \( fr := rf^{-1} \circ co \);
- the communication relation gathers reads-from, coherence and from-reads: \( \text{com} := rf \cup co \cup fr \);
- the dependency relation gathers address and data (but not control) dependencies: \( \text{dep} := addr \cup data \);
- the program order relation restricted to accesses of the same location: \( po-loc := po \cap loc \);
- the internal reads-from relation, i.e., the reads-from which take place within a thread: \( rf_i := rf \cap int \);
- the external relation \( ext \), containing pairs of events that belong to different threads: \( ext := \sim \cap int \);
- the external reads-from, coherence and from-reads: \( \text{rfe} := rf \cap ext \circ co := co \cap ext \circ rf \cap ext \).

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 (\( fr \)) 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 \( T_i \) 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.

\[
\begin{align*}
\text{acyclic}(po-loc \cup \text{com}) & \quad \text{(Scpv)} \\
\text{empty}(rmw \cap (fre \circ coe)) & \quad \text{(At)} \\
\text{acyclic}(hb) & \quad \text{(Hb)} \\
\text{acyclic}(pb) & \quad \text{(Pb)}
\end{align*}
\]

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 eq-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. \texttt{rfi-rel-acq} is an internal reads-from communication between a release and an acquire. The \texttt{rm} relation pairs reads with an \texttt{smp\_rm} fence between them. Similarly, \texttt{wmb} pairs reads with an \texttt{smp\_wmb} fence between them, \texttt{mb} pairs any events with an \texttt{smp\_mb} fence between them, and \texttt{rb-dep} pairs reads with a \texttt{smp\_read\_barrier\_depends} between them.

3.1 Examples

\texttt{LB+ctrl+mb}, in Figure 4, appears in the ring-buffer interface from kernel to userspace (see \texttt{perf\_output\_put\_handle()} in [82, kernel/events/ring_buffer.c]). \texttt{T_0} reads from \texttt{x} (event \texttt{a}) and writes to \texttt{y} (\texttt{b}), imposing a control dependency (depicted by the \texttt{ctrl} arrow) in between. Similarly, \texttt{T_1} reads from \texttt{y} (\texttt{c}) and writes to \texttt{x} (\texttt{d}), with an \texttt{smp\_mb} fence between them (\texttt{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].

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

\texttt{SB+mb}s (a store buffering idiom), in Figure 6, is used in LK wait-event/wakeup code. It is documented in functions \texttt{wait\_active()} [82, include/linux/wait.h]; \texttt{wait\_woken()} and \texttt{woken\_wake\_function()} [82,kernel/sched/wait.c]; and \texttt{wake\_q\_add()}, \texttt{wake\_up\_q()}, and \texttt{try\_to\_wake\_up()} [82,kernel/sched/core.c]. Without the fences it is observed on x86.

\texttt{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. \texttt{SB+mb}s: Forbidden.

Figure 7. \texttt{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.

dep := addr \cup data

\texttt{rdep} := (\texttt{addr} \cup \texttt{ctrl}) \cap (R \times W)

\texttt{overwrite} := \texttt{co} \cup \texttt{fr}

to-\texttt{w} := \texttt{rdep} \cup (\texttt{overwrite} \cap \texttt{int})

\texttt{rrdep} := \texttt{addr} \cup (\texttt{dep} \cup \texttt{rfi})

\texttt{strong-rrdep} := \texttt{rrdep} \cap \texttt{rb-dep}

to-r := \texttt{strong-rrdep} \cup \texttt{rfi-rel-acq}

\texttt{strong-fence} := \texttt{mb}

\texttt{fence} := \texttt{strong-fence} \cup \texttt{po-rel} \cup \texttt{wmb} \cup \texttt{rmb} \cup \texttt{acq-po}

\texttt{ppo} := \texttt{rrdep} \cap (\texttt{to-r} \cap \texttt{to-w} \cup \texttt{fence})

\texttt{cumul-fence} := \texttt{A-cumul(strong-fence} \cup \texttt{po-rel}) \cup \texttt{wmb}

\texttt{prop} := (\texttt{overwrite} \cap \texttt{ext})\cup (\texttt{cumul-fence}\cup \texttt{rfe})

\texttt{hb} := (\texttt{prop} \cap \texttt{id}) \cup \texttt{int} \cup \texttt{ppo} \cup \texttt{rfe}

\texttt{pb} := \texttt{prop} \cup \texttt{strong-fence} \cup \texttt{hb}

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 \texttt{READ\_ONE}. 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 \texttt{ppo}

\texttt{ppo} relates events in program order as described below:

Local orderings to writes are modeled by \texttt{to-w}. The \texttt{rdep} (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 (acc-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.

**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 T₁ is a write release, thus (b, c) ∈ po-rel. Since b reads the write a in T₀, (c, 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 ϵ₁ and ϵ₂ are related by a sequence of cumul-fence links:

- (ϵ₁, ϵ₂) ∈ 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 ϵ₁ links by prop to ϵ₂. In Figure 2, d is overwritten by a; thus (d, b) ∈ prop.
- ϵ₁ (or an external event it overwrites) is related by prop to events that read from ϵ₂. In Figure 7, b is overwritten by c and the release d is read by e; thus (b, e) ∈ prop.
- These facts hold when ϵ₁ = ϵ₂. For instance, in Figure 6, d is overwritten by a; thus (d, a) ∈ prop. Ident 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 → b, rfe → c, ppo → d, 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 ∩ d) ∩ int. And since d and e are separated by an rmb fence, we also have (d, e) ∈ ppo. Thus d → ppo e → (prop ∩ d) ∩ 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.

**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) ∈ acc-po, which entails (d, e) ∈ strong-rrdep ≤ to-r. Therefore (c, e) ∈ ppo.
In Figure 6, \((d, a) \in \text{prop}\), as mentioned above. Since \(a\) and \(b\) are separated by a strong fence, we have \((d, b) \in \text{pb}\). By symmetry we also have \((b, d) \in \text{pb}\), hence a cycle in \(\text{pb}\).

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

### 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 \(\text{ppo}\), which models local orderings due to dependencies (to-r and to-w) and fences;
- the propagation relation \(\text{prop}\), 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 \(\text{hb}\), which models the effect of local orderings due to ppo and fences on reads-from;
- the propagates-before relation \(\text{pb}\), modeling strong fences.

### 4 Modeling Read-Copy-Update

#### Table 4. RCU primitives and corresponding events

<table>
<thead>
<tr>
<th>LK/C primitive</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcu_dereference()</td>
<td>R[once],F[rb-dep]</td>
</tr>
<tr>
<td>rcu_assign_pointer()</td>
<td>W[release]</td>
</tr>
<tr>
<td>rcu_read_lock()</td>
<td>F[rcu-lock]</td>
</tr>
<tr>
<td>rcu_read_unlock()</td>
<td>F[rcu-unlock]</td>
</tr>
<tr>
<td>synchronize_rcu()</td>
<td>F[sync-rcu]</td>
</tr>
</tbody>
</table>

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 \(\text{rcu}_\text{read_lock}\) and \(\text{rcu}_\text{read_unlock}\) to delimit a \textit{read-side critical section} (RSCS). \textit{Updaters} are writers that call the \(\text{synchronize}_\text{rcu}\) primitive; calling it starts a \textit{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 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:

\textit{RCU’s specification is primarily empirical in nature}

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

\textbf{Read-side critical sections cannot span grace periods}.

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

- \textbf{RSCS precedes GP}: if any access in the RSCS precedes the GP, then no access in the RSCS can follow the GP.
- \textbf{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 \(rf\) arrow to indicate that \(b\) precedes \(c\), hence \(b\) precedes the \(\text{synchronize}_\text{rcu}\) 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 \(d\) follows \(a\), i.e., \(a\) executes after \(\text{synchronize}_\text{rcu}\) 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 \(\text{wmb}\) in \(T_1\) and \(\text{rmb}\) 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:

\[ F(\text{RSCS}, \text{GP}) = \text{RSCS} \quad \text{or} \quad F(\text{RSCS}, \text{GP}) = \text{GP}. \]
We augment our model with the relations in Figure 12.

order separated by a synchronize_rcu

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. $F(RSCS,GP) = RSCS \land (e_1,u) \in po \land (s,e_2) \in po^*$
- 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. $F(RSCS,GP) = GP \land (e_1,s) \in po \land (l,e_2) \in po^*$

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

- If $F(RSCS,GP) = RSCS$ (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 $(a,d) \in rcu-fence(F)$.
- If $F(RSCS,GP) = GP$ (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 $(c,b) \in rcu-fence(F)$.

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:

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

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 $F(RSCS,GP) = RSCS$ then $(a,d) \in rcu-fence(F)$. Moreover we have $(d,a) \in rfe$, thus in $hb^*$. This creates a cycle in $pb(F)$.
- if $F(RSCS,GP) = GP$ then $(c,b) \in rcu-fence(F)$. Moreover we have $(b,c) \in rfe$, 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 \cap (_ \times Sync)) \cap po^*
strong-fence := mb \cup gp
rscs := po \cup crit^{-1} \cup po'
lk := hb^* \cup prop

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, $(g,g)$, $(g,a)$, $(a,b)$, $(b,a)$, $(b,z)$, $(b,g)$, 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. As such, it is acyclic.

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.
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.
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 1d, 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 \subseteq to-w \subseteq 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.
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 \( (e_1, e_2) \in \text{gp} \) in \( X \) (e.g., the writes \( c \) and \( d \) in Figure 10). There is an \( F[\text{sync-rcu}] \) event between them in program order; hence the \( F[\text{mb}] \) event arising from line 44 lies between the corresponding events \( e'_1 \) and \( e'_2 \) in \( X' \). Thus \( (e'_1, e'_2) \in \text{mb} \), implying that \( (e'_1, e'_2) \in \text{strong-fence} \). (Between \( c' \) and \( d' \) in Figure 16 are all the events from Figure 15’s implementation of \text{sync-rcu}; the \( F[\text{mb}] \) 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 \( \text{pb}(F) \) 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 \( \text{rc}[i] \) executed by line 27 of Figure 15,
- in the call to \text{gp}_ongoing(i) from the last iteration of the while loop at line 38,
- in the first call to \text{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 \( \text{rcu} \_\text{read} \_\text{lock} \) was not visible at the start of the GP: \( (r_1,l') \in \text{fr} \);
2. the RSCS’s \( \text{rcu} \_\text{read} \_\text{unlock} \) or a later write to \( \text{rc} \[1 \] \) was visible at the end of the GP: \( (u',r_2) \in (\text{coi}\overline{1};\text{rf}) \).

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

A cycle in \( \text{pb}(F) \) for \( X \) would give rise to a cycle in \( \text{pb} \) 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 \( \text{pb}(F) \) relation for \( X \) contains a cycle. We are now claiming this means that \( X' \) in Figure 16 has a cycle in \( \text{pb} \). And so it does: \( d' \xrightarrow{\text{rc} \_\text{f}} a' \xrightarrow{\text{nb}} j' \), hence \( d' \xrightarrow{\text{pb}} j' \), and similarly, \( j' \xrightarrow{\text{pb}} d' \text{ via } m' \).

Returning to the general proof of Theorem 2: The theorem assumed that \( X' \) is allowed in our model and hence obeys the \( \text{pb} \) constraint. This requires the \( \text{pb} \) relation in \( X' \) to be acyclic, from which we now deduce that the \( \text{pb}(F) \) 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, \( \text{smp} \_\text{read} \_\text{barrier} \_\text{depends} \) exists exclusively for the sake of Alpha. Otherwise, in the definition of \( \text{ppo} \), the relations \( \text{strong-\text{r}dep} \) and \( \text{r}dep \) 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 \( \text{spin} \_\text{lock} \) primitive behaves like \( \text{xchg} \_\text{acquire} \) for this location. In Table 3, this is modeled as a read with annotation acquire and a write with annotation once, governed by the \( \text{At} \) axiom of Figure 3 and the constraints on acquire in Figure 8. The \( \text{spin} \_\text{unlock} \) primitive behaves like a \( \text{smp} \_\text{store} \_\text{release} \) 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 \( \text{call} \_\text{rcu} \) and \( \text{rcu} \_\text{barrier} \).

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
