Formalizing the User-Kernel Shared-Memory Boundary: The C/LKMM Compound Model

Hastin Kapoor
University of Utah
Salt Lake City, UT, USA
u1528432@utah.edu

Soham Bagchi
University of Utah
Salt Lake City, UT, USA
soham.bagchi@utah.edu

An Qi Zhang
University of Utah
Salt Lake City, UT, USA
an.qi.zhang@utah.edu

Soham Chakraborty
TU Delft
Delft, Netherlands
s.s.chakraborty@tudelft.nl

Marco Elver
Google
Munich, Germany
melver@kernel.org

Andrés Goens
TU Darmstadt
Darmstadt, Germany
andres.goens@tu-darmstadt.de

Susmit Sarkar
University of St Andrews
St Andrews, United Kingdom
Susmit.Sarkar@st-andrews.ac.uk

Ryan Stutsman
University of Utah
Salt Lake City, UT, USA
stutsman@cs.utah.edu

Vijay Nagarajan
University of Utah
Salt Lake City, UT, USA
vijay@cs.utah.edu

Abstract

The boundary between an operating-system kernel and its user-space applications has moved into shared memory. Zero-copy interfaces such as `io_uring` and BPF arenas place shared memory directly between code that obeys the C memory model and code that obeys the Linux Kernel Memory Model (LKMM). No formal model spans this boundary, and the two underlying models are structurally different. C builds happens-before from explicit release-acquire pairs, while the LKMM builds it from preserved program order.

We present the C/LKMM compound memory model, the first formal memory model specification of the user-kernel boundary. The compound model takes the LKMM as its structural basis and bridges it with C through two ideas: a *synchronize-threads* relation that parameterizes synchronization strength by the language of the operations involved, and *fence context sharing*, which lets a single LKMM fence context participate in multiple synchronizations without leaving its native context. We prove three correctness properties: pure-C programs retain their C semantics, pure-Linux programs retain their LKMM semantics, and shared synchronization primitives are interchangeable across the boundary—and complement them with systematic litmus-test validation. Using the model as a guide, we develop a family of lock-free data structures shared between user-space C and eBPF kernel code over BPF arenas. Finally, we identify and fix a real cross-boundary memory ordering bug in the Linux kernel's `kcov` subsystem; both halves of the fix have been accepted upstream.

CCS Concepts: • **Theory of computation** → **Concurrency**; • **Software and its engineering** → *Operating systems.*

Keywords: memory consistency models; LKMM; compound memory model; eBPF

ACM Reference Format:
Hastin Kapoor, Soham Bagchi, An Qi Zhang, Soham Chakraborty, Marco Elver, Andrés Goens, Susmit Sarkar, Ryan Stutsman, and Vijay Nagarajan. 2027. Formalizing the User-Kernel Shared-Memory Boundary: The C/LKMM Compound Model. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3845814.3850055

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850055

1 Introduction

The boundary between an operating system kernel and user-space applications is undergoing a fundamental shift. To meet the demands of modern hardware, user-space programs and the kernel increasingly share data directly, synchronizing in a zero-copy fashion using interfaces like `io_uring` and eBPF (extended Berkeley Packet Filter). This intimacy raises a key question: what defines correctness in these user-kernel interactions? A formal, unified memory model that describes this boundary is conspicuously absent.

Correctness for user-space is defined by the C memory model [12, 15], while the kernel adheres to the Linux Kernel Memory Model (LKMM) [2, 17]. Each has evolved to serve its domain. C allows programmers to explicitly request the ordering they need while leaving the compiler and hardware free to optimize. The LKMM codifies the ordering that Linux-supported hardware provides. Dependency ordering is the

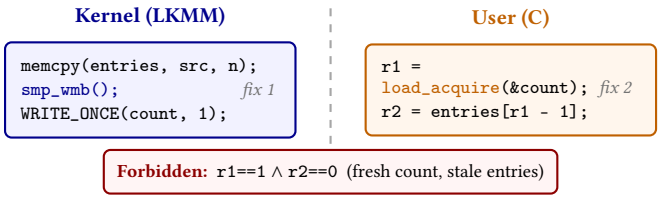

Figure 1. The `kcov_remote` cross-boundary bug. Initially `count = 0` and `entries[]` is zeroed; `src` holds nonzero data. The figure shows the first append, so the count update writes 1. `load_acquire` abbreviates `__atomic_load_n(..., __ATOMIC_ACQUIRE)`. Neither fix alone suffices: `smp_wmb()` on the kernel side has no effect if the user-side load is C-relaxed, and an acquire load on the user side has no effect if the kernel's stores are unordered under LKMM. The bug is closed only by a matched pair across the boundary—precisely the contract of our C/LKMM compound memory model.

clearest divide: hardware honors it, but C cannot promise it, since a compiler may optimize the dependency away. Thus, in C, cross-thread ordering is governed almost exclusively by release-acquire pairs; it exists only where the programmer explicitly asks for it. The LKMM, in contrast, relies on *preserved program order* (ppo), the base of its ordering hierarchy: a designated set of intra-thread patterns—address, data, and control dependencies, and fences—that the model trusts the hardware, and the compiler for guideline-conforming code, to preserve. Cross-thread ordering then arises whenever one thread reads a value written by another, linking the writer's ppo to the reader's ppo; no explicit release or acquire is required. It is at this level that the two models structurally diverge; the LKMM's stronger mechanisms—cumulative fences and full barriers—map more directly onto C's fences (Section 2). The address-dependency case is a key example: in the LKMM, a pointer read followed by its dereference is automatically ordered, and this is a standard kernel synchronization pattern; in C, the same code orders nothing unless the pointer read is tagged `acquire`. A similar difference shows up with fences. C treats fences as independent synchronization points, while the LKMM defines each fence only in context with the surrounding memory operations it orders, a design that reflects how fences compile to hardware.

Each model serves its domain well in isolation, but the two now meet directly on shared memory, and intuitions carried over from one domain can lead to subtle, hard-to-diagnose concurrency bugs in code where user-space and kernel interact.

A concrete example illustrates the problem (Figure 1). The Linux kernel's `kcov_remote` shares a coverage buffer with a user-space consumer: the buffer's first word holds the number of valid entries, and the remaining words hold the entries themselves. The kernel appends by `memcpy`-ing new entries into the buffer and then updating the count; user space loads the count and then reads that many entries. Until recently, the kernel-side producer omitted a write barrier between the `memcpy` and the count update, and the user-side consumer loaded the count with a C relaxed atomic. Each omission is a missing ordering under its own model: the kernel stores are unordered under LKMM without an explicit barrier, and C neither orders reads after a relaxed load nor honors the address dependency from the count to the entry read. Both omissions sit at the kernel-user interface, and we read that as no coincidence but as a symptom of a missing contract: with no formal statement of what the boundary guarantees, it is natural for a programmer on either side to assume—incorrectly—that the other side, or the interface itself, supplies the ordering. Closing the bug requires a matched pair, `smp_wmb()` on the producer and an acquire load on the consumer, and neither alone suffices—the kernel barrier orders its own stores but cannot force a relaxed C load to observe them in order, and an acquire load on the user side has nothing to synchronize against if the kernel's stores remain unordered. We identified the bug, proposed both fixes, and both patches have been accepted upstream. This matched pair—publication ordering on the kernel side, an acquire on the user side—is precisely the contract our compound memory model expresses.

We argue that this boundary must be placed on a solid formal foundation. This foundation must reflect the intuition that the core synchronization primitives of each domain should be semantically *interchangeable*, allowing a C release to correctly synchronize with a kernel acquire, and vice versa. To this end, we present the C/LKMM compound memory model, the first formal specification of the user-kernel memory boundary. Our compound model preserves the exact semantics of pure C and pure Linux programs—neither model is modified in isolation—and unifies them by extending both the happens-before and the sequentially-consistent (SC) axioms across the boundary, so that releases, acquires, and SC fences remain semantically interchangeable between the two memory models.

The compound model takes the LKMM as its structural basis and extends it with two ideas that together deliver interchangeability. First, we introduce a *synchronize-threads* relation (`st`) that parametrizes synchronization strength by the language of the operations involved: a cross-thread synchronization involving C operations is held to C's release-acquire requirement, while one between LKMM operations follows LKMM's native ppo rules. Second, we introduce *fence context sharing*: we treat each LKMM fence together with its required surrounding operations as a single inseparable unit, and allow the same unit to participate in multiple synchronizations at once—for example, as the acquire-side of one synchronization and the release-side of another—without leaving its native context. Our model covers all the fences and synchronization primitives of both C and LKMM, except

C's per-access SC atomics, which have no kernel counterpart; SC fences are fully supported, corresponding to the kernel's `smp_mb()` (Section 3).

We formally prove three correctness properties of the compound model: pure-C programs retain their C semantics, pure-Linux programs retain their LKMM semantics, and the shared synchronization primitives are interchangeable across the boundary. We complement these proofs with systematic litmus testing. For the preservation properties, we run thousands of existing C and LKMM litmus tests under both their native model and the compound model, confirming that the allowed behaviors match in each case. For interchangeability, we take the same tests and systematically substitute each shared synchronization primitive with its counterpart in the other language, confirming that the allowed behaviors remain unchanged.

To demonstrate the model's utility, we develop a family of lock-free concurrent data structures shared between user-space C code and eBPF kernel code. The BPF memory model is under active formalization [6, 18]; we treat eBPF kernel code as LKMM-governed, consistent with the proposed model on the synchronization operations our case studies use. Each data structure runs on BPF Arenas, which provide zero-syscall shared memory between user space and the kernel. We apply the compound model to annotate producer and consumer paths separately for C and LKMM, and reason about heterogeneous synchronization across the boundary. The resulting implementations are both worked examples of compound-model reasoning and standalone contributions to user-kernel concurrent programming.

Contributions.

- The *C/LKMM compound memory model*, the first formal specification of the user-kernel memory boundary.
- Formal proofs of three correctness properties—pure-C programs retain their C semantics, pure-Linux programs retain their LKMM semantics, and the shared synchronization primitives are interchangeable across the boundary—complemented by systematic litmus-test validation.
- A family of lock-free concurrent data structures shared between user-space C code and eBPF kernel code over BPF arenas—including the Michael-Scott queue, ring buffers, and stacks—that serve as worked case studies in compound-model reasoning and are of standalone interest for user-kernel concurrent programming.
- A real-world cross-boundary bug, identified using the compound model as a formal guide: a memory ordering bug in the Linux kernel's `kcov` subsystem that requires a matched pair of fixes—a write barrier on the kernel producer and an acquire load on the user consumer—both of which have been accepted upstream.

Relation	Name	Definition
W, R, F	—	write, read, fence
po / sb	program order	intra-thread order
rf	reads from	relates a write to a read that reads its value
co / mo	coherence order	order of writes to a location
fr	from read	relates read to a later write, $\mathsf{rf}^{-1};\mathsf{co}$
rfe	external reads-from	$\mathsf{rf} \cap \mathsf{ext}$
com	communication	$\mathsf{rf} \mid \mathsf{co} \mid \mathsf{fr}$
eco	extended communication	com^+
sw	synchronizes with	release-acquire sync
ppo	preserved program order	subset of po that is ordered
prop	propagation	propagation across threads via cumulative fences (§2.2)
Marked	—	Linux access via a kernel accessor (`READ_ONCE()`, …)
Plain	—	any other access; in our setting, every C access
$r^?$, r^+, r^*	—	reflexive, transitive, and reflexive-transitive closure

$r_1;r_2$ is composition, $r_1 \mid r_2$ is union. Composition binds more tightly than union. The suffix e denotes an external relation, i.e. with endpoints in distinct threads, and i denotes internal. $[X]$ is the identity relation restricted to events in class X (e.g., $[\mathsf{W}]$, the identity on writes). In the model-summary tables, $\subset$ marks abbreviated definitions: we give only a bound on the relation, not its full form. The relation loc relates accesses to the same location, and the subscript $\neq$ loc restricts a relation to pairs on different locations. Throughout the paper, colors indicate a relation's provenance: shared base relations, C, LKMM, and compound-model additions. Marked accesses in LKMM play a role analogous to that of C's atomic accesses.

Table 1. Core definitions used in each model.

2 Background

The C [12, 15] and Linux Kernel Memory Model (LKMM) [2, 17] differ fundamentally in design philosophy; we give a brief primer on each below and refer readers to the cited works for full descriptions. Table 1 summarizes the notation used throughout the paper.

2.1 The C Memory Model

The C memory model builds cross-thread ordering from explicit synchronization primitives; Table 2 summarizes the model. The core relation is "synchronizes with" (sw), created in the common case by a release-acquire pair: when an acquire read observes a release write, everything sequenced before the release becomes visible to everything sequenced after the acquire.

Figure 2 shows the standard message-passing pattern. The release-acquire pair creates a sw edge from the flag write to the flag read. Combined with sequenced-before (sb) on each thread, this gives a "happens-before" (hb_C) path from the data write to the data read. C enforces ordering by requiring irreflexivity of hb_C composed with communication: if the

data read were to see the stale value, a from-read (fr) edge would close a cycle, and the coherence axiom would forbid it.

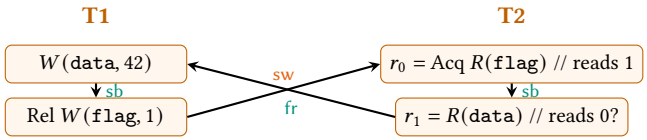

Figure 2. Message passing in C. The release-acquire pair creates an sw edge. If the data read sees the stale value, fr closes a cycle with $\mathsf{hb_C}$, forbidden by C's coherence axiom.

Using the definition of C's coherence axiom, we see the cycle in Figure 2—$\mathsf{hb_C}$ from the data write to the data read, then fr back—is forbidden.

$$\mathsf{hb_C} = (\mathsf{po}|\mathsf{sw})^+$$

$$\text{irreflexive } (\mathsf{hb_C} ; \mathsf{eco}^?)$$

The full definition of sw also includes "release-sequences" (rs): a chain of zero or more rmws that read from the release, with the acquire reading from the last of them. As of C++20 [5], release sequences no longer include writes of the releasing thread that follow the release in program order; we adopt this stricter definition.

A key point: co and fr never enter $\mathsf{hb_C}$ itself—C builds $\mathsf{hb_C}$ exclusively from program order and release-acquire synchronization, and uses eco only to check that this ordering is consistent with communication. LKMM, as we shall see, takes a different approach.

C's second axiom, PSC, governs SC fences and atomics. The full axiom requires acyclic (pscb | pscf), but since we omit SC accesses [15] (justified in Section 3), only pscf is relevant:

$$\mathsf{pscf} = [\mathsf{F} \cap \mathsf{sc}] ; (\mathsf{hb_C} \mid \mathsf{hb_C} ; \mathsf{eco} ; \mathsf{hb_C}) ; [\mathsf{F} \cap \mathsf{sc}]$$

The endpoints of pscf are the fences themselves—a structural difference from the LKMM, where fences must remain embedded in the context of their surrounding operations.

2.2 The Linux Kernel Memory Model

In the LKMM, the picture inverts; Table 3 summarizes the model. C admits only release-acquire across threads, yet preserves all of program order within a thread. The LKMM does the opposite. Across threads it is permissive: any cross-thread read-from (rfe) synchronizes, with no release or acquire needed. Within a thread it is selective: only address, data, and control dependencies and fences are preserved—a subset called *preserved program order* (ppo). This inversion is by design: the LKMM behaves like a hardware memory model, abstracting the machines Linux runs on. Cumulative fences and full barriers, introduced below, provide successively stronger ordering atop this base. Accesses made through the kernel's accessors (`READ_ONCE()`, `WRITE_ONCE()`, ...)

Relation / Axiom	Definition
rs	$[\mathsf{W}] ; (\mathsf{rf} ; \mathsf{rmw})^*$
sw	$([\mathsf{W}^{\geq \mathsf{rel}}] \mid [\mathsf{F}^{\geq \mathsf{rel}}] ; \mathsf{po} ; [\mathsf{W}]) ; \mathsf{rs} ; \mathsf{rf}$; $([\mathsf{R}^{\geq \mathsf{acq}}] \mid [\mathsf{R}] ; \mathsf{po} ; [\mathsf{F}^{\geq \mathsf{acq}}])$
$\mathsf{hb_C}$	$(\mathsf{po}\|\mathsf{sw})^+$
scb	$\mathsf{po} \mid (\mathsf{po}_{\neq \mathsf{loc}} ; \mathsf{hb_C} ; \mathsf{po}_{\neq \mathsf{loc}}) \mid (\mathsf{hb_C} \cap \mathsf{loc}) \mid \mathsf{co} \mid \mathsf{fr}$
pscb	$([\mathsf{SC}] \mid [\mathsf{F} \cap \mathsf{SC}] ; \mathsf{hb_C}^?) ; \mathsf{scb}$; $(\mathsf{hb_C}^? ; [\mathsf{F} \cap \mathsf{SC}] \mid [\mathsf{SC}])$
pscf	$[\mathsf{F} \cap \mathsf{SC}] ; (\mathsf{hb_C} \mid \mathsf{hb_C} ; \mathsf{eco} ; \mathsf{hb_C}) ; [\mathsf{F} \cap \mathsf{SC}]$
coherence	irreflexive $(\mathsf{hb_C} ; \mathsf{eco}^?)$
PSC	acyclic (pscb | pscf)
atomicity	empty $(\mathsf{rmw} \cap (\mathsf{fr} ; \mathsf{co}))$
no-thin-air	acyclic (po | rf)

Table 2. Summarized C memory model

are called *Marked*; unannotated accesses are *Plain*. Note that while LKMM preserves dependencies, those survive compilation only for code that follows the kernel's coding guidelines [10]; the LKMM assumes this discipline, and our compound model inherits the assumption.

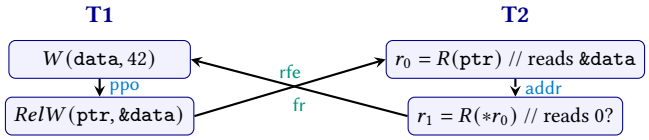

Figure 3. Address-dependency ordering in the LKMM. The address dependency from $R(\texttt{ptr})$ to $R(*r_0)$ is ppo; combined with the LKMM hb ordering from $R(*r_0)$ to $R(ptr)$, this forbids the stale read. In C, the same code provides no ordering; $R(\texttt{ptr})$ would need an acquire annotation.

Figure 3 illustrates this: the address dependency from a pointer read to its dereference is ppo, giving the LKMM an ordering guarantee that C does not provide without an explicit acquire.

The LKMM's happens-before has a third component beyond ppo and rfe: the *propagation* relation (prop). True to its hardware lineage, the LKMM derives happens-before from inter-thread co and fr edges in the presence of cumulative fences (cumul-fence): the prop relation chains an optional external co or fr edge with cumul-fence orderings—where C keeps co and fr outside $\mathsf{hb_C}$ entirely. The resulting definition is:

$$\mathsf{hb} = \mathsf{ppo} \mid \mathsf{rfe} \mid (\mathsf{prop} \setminus \mathsf{id}) \cap \mathsf{int}$$

Unlike C's happens-before, this is not a transitive closure—each component contributes directly, and the $\setminus$ *id* restriction prevents trivial self-ordering.

The LKMM's counterpart to pscf is *propagates-before* (pb), but its structure differs because the LKMM treats fences as meaningful only in the context of the operations they order. For example, a `Before_atomic` fence provides ordering

guarantees only when followed by an atomic RMW; outside this context it has no meaning, which reflects how it compiles to hardware—on x86, `Before_atomic` compiles to a no-op since it would be redundant against the strong ordering guarantees of the following RMW. Due to this requirement, fences cannot be the endpoints of the relation as they are in C's pscf.

The LKMM defines the propagates-before relation pb = prop ; strong-fence ; hb* and requires acyclic pb, keeping the fence in the center of its context. This structural difference—fences as endpoints in C versus fences in context in the LKMM—is one of the core challenges the compound model must reconcile (Section 3).

Relation / Axiom	Definition
ppo	⊂ po
cumul-fence	⊂ ppo, generally ≥ Rel
strong-fence	⊂ ppo, generally ≥ SC
prop	⊂ (coe ∣ fre)? ; (rfe ; cumul-fence)* ; rfe?
hb	ppo ∣ rfe ∣ (prop \ id) ∩ int
pb	prop ; strong-fence ; hb*
happens-before	acyclic(hb)
propagation	acyclic(pb)
atomicity	empty (rmw ∩ (fre ; coe))
SC per Location	acyclic ($po_{=loc}$ ∣ com)

Table 3. Summarized Linux kernel memory model

2.3 Comparing C and LKMM

Table 4 highlights some of these key differences in design.

	C	LKMM
Inter-Thread	Rel-Acq	Any reads-from (rfe)
Intra-Thread	Prog. Order (po)	Preserved (ppo)
Non-rfe Comm.	Outside *hb* (eco)	Inside *hb* (prop)
Fences	Standalone Sync	Requires Context
SC Axiom	Endpoints (pscf)	Central (pb)
Philosophy	Portable Abstr.	Hardware-Centric

Table 4. Summary of Differences between C and LKMM

3 Overview

3.1 Challenges

Constructing a compound model is nontrivial due to the structural differences outlined above. Two problems arise.

Problem 1: Defining Synchronization. The two models cannot simply be unioned. Starting from C and adding LKMM's preserved program orders does not work: *ppo* is already a subset of program order, so C absorbs it with no effect, while C's requirement for release-acquire synchronization remains overly strict for Linux code. Starting from the LKMM is more promising, but raises its own questions: C depends on *all* program-order edges being preserved, which conflicts with the LKMM's restricted *ppo*, and the LKMM's permissive inter-thread synchronization—any *rfe* suffices—conflicts with C's requirement that cross-thread ordering be mediated by release-acquire pairs.

Problem 2: Matching the Fence Requirements of C and Linux. C and the LKMM treat fences in fundamentally different ways. In C, a fence is a standalone synchronization point: in Figure 4, the SC fence acts as an acquire to synchronize with T1's release, and as a release to synchronize with T3's acquire—participating in two *sw* edges simultaneously.

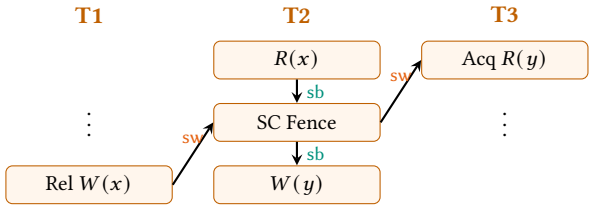

Figure 4. SC fence synchronization in C. The SC fence participates in two synchronizations: as an acquire (synchronizing with T1's release via sw) and as a release (synchronizing with T3's acquire via sw). In the LKMM, the fence has no standalone meaning—it is only meaningful in the context of the surrounding operations $R(x)$; SC Fence ; $W(y)$.

In LKMM, a fence has no meaning on its own—it is a pattern-matching trigger whose semantics depend on the operations that surround it. A `Before_atomic` fence, for example, provides ordering only when followed by an atomic RMW; without this context, it compiles to a no-op on x86 (Section 2.2).

This difference creates a dilemma for the compound model. Following C's approach and treating fences as standalone endpoints would be unsound for LKMM fences. A `Before_atomic` not followed by an RMW would be credited with ordering it does not provide. Following the LKMM's approach—requiring fences to remain embedded in their surrounding context—breaks C's synchronization patterns. In Figure 4, both $R(x)$ and $W(y)$ are relaxed; only the SC fence provides acquire and release behavior. If the compound model does not allow fences to participate in synchronization, neither cross-thread edge can be established, and the model would allow T3 to read stale data—a behavior C forbids. A new approach is needed that respects the LKMM's context requirement while recovering C's fence synchronization patterns.

3.2 Approach

We take the LKMM as the structural basis for the compound model, with two modifications.

Synchronize-threads (st). We replace rfe in the LKMM's happens-before with an abstract relation, st, that parametrizes synchronization requirements by language. Between two

LKMM operations, st requires only rfe—the LKMM default. When both operations are C, st requires a release on the writer and an acquire on the reader, matching C's sw requirements. In the cross-boundary case, each side contributes only what its own language demands: a kernel write need not be a release to synchronize with a C acquire, and a kernel read need not be an acquire to synchronize with a C release.

To recover C's intra-thread ordering, we extend ppo by generalizing the LKMM's *po-rel* and *acq-po*: any operation before a release, or after an acquire, is preserved. These are the implicit program orders that C's sw-based happens-before relies on. Additional cases for synchronization in C, such as intervening RMWs between release and acquire, are encoded in st without impacting Linux behavior.

Fence context sharing. To resolve Problem 2, we treat each fence together with its surrounding operations as a unit and allow the unit to participate in multiple synchronizations. In Figure 4, the release on T1 synchronizes not with the SC fence alone but with the entire context $R(x)$; SC Fence ; $W(y)$, and this same context synchronizes with the acquire on T3. This preserves the LKMM's requirement that fences remain embedded in context—ensuring that pattern-dependent fences like `Before_atomic` are never credited with ordering outside their context—while recovering C's ability to route multiple synchronizations through a single fence.

SC accesses. C provides two sequentially consistent (SC) primitives: SC fences and SC accesses. The LKMM provides only the former: the kernel orders through the full barrier `smp_mb()`, to which C's `atomic_thread_fence(seq_cst)` corresponds—both are strong fences in the compound model. A per-access SC tag has no kernel counterpart. The compound model therefore gives cross-boundary SC ordering semantics through SC fences alone. In purely C programs, SC accesses retain their usual semantics.

4 A Compound Model

We take the LKMM as the structural basis and extend it in two steps: §4.1 introduces the main changes to synchronization and preserved program order; §4.2 develops the generalized relation composition needed to reconcile C and LKMM fence semantics; §4.3 gives the complete definition.

4.1 Major Changes From the LKMM

The main changes to the LKMM for building the compound memory model relate to the happens-before relation, which in LKMM is defined as:

$$\text{hb} = \text{ppo} \mid \text{rfe} \mid (\text{prop} \setminus \text{id}) \cap \text{int}$$

In the following, we discuss changes both to the relations defining it and the corresponding axiom. The definitions below compose relations with ⨟ rather than ordinary composition. This is the formal vehicle for the fence context sharing of §3: ordinary composition joins two relations at a single shared event, whereas ⨟ also allows them to overlap on an enclosing fence context, so that a fence, together with its context, can participate in two synchronizations at once. § 4.2 gives the formal definition. Numbered equations cited in this subsection appear in the full model definition (§ 4.3).

Extended preserved program order. C's happens-before includes all program orders (po), but the compound model inherits the LKMM's restricted ppo which preserves only specific intra-thread patterns. However, C's guarantees rest only on the program orders adjacent to its synchronization—the orders implicit in §2's release-acquire guarantee: everything before a release stays before it, everything after an acquire stays after it, and likewise around a fence. We therefore extend ppo with exactly these, by extending the LKMM's po-rel (order into a release), acq-po (order out of an acquire), and strong-fence (covering C's SC fence), and adding weak-fence for C's acquire-release fence (Eqns. 5–6).

Synchronize-threads (st). The inter-thread side is where the two models diverge more: the LKMM synchronizes on any rfe, C only on a release-acquire pair (the inversion of §2), and a compound program must respect both at once. We therefore introduce *synchronize-threads* (st, Eqn. 9), which parametrizes the synchronization requirement by language: it reduces to Linux's weaker synchronization requirements on Linux programs while matching C's stronger sw on C programs.

$$\text{st} = \text{inter-start} \fatsemi (\text{rfe} \cup (\text{rf} \fatsemi \text{rmw})^{+} \fatsemi \text{rf}^{?}) \fatsemi \text{inter-end}$$

inter-start and inter-end (Eqn. 7 and Eqn. 8) restrict how a synchronization may start and end. inter-start is satisfied if the synchronization starts with a Marked access (Table 1)—even one carrying no ordering annotation, such as a kernel `WRITE_ONCE()`—or with an event or sequence at least as strong as a release. Similarly, inter-end is satisfied if the synchronization ends with a Marked access—even an unordered `READ_ONCE()`—or with an event or sequence at least as strong as an acquire.

This recovers the behavior of sw: C accesses cannot satisfy the Marked requirement and thus must be releases and acquires, or sequences that provide at least the same guarantees as releases and acquires. It doesn't restrict synchronization in Linux to using release and acquire, since Marked accesses satisfy the predicates that restrict the start and end of a synchronization trivially—no annotation is needed. In the cross-boundary case, each side is checked independently: a kernel write need not be a release to synchronize with a C acquire, and vice versa.

Between these endpoints, the middle term $(\text{rfe} \cup (\text{rf} \fatsemi \text{rmw})^{+} \fatsemi \text{rf}^{?})$ allows intervening RMWs between release and acquire, matching C's treatment of release-acquire chains. Notably, while this definition is more permissive than using rfe, it does not conflict with the notion of happens before in the LKMM: any such sequence of writes, RMWs and reads can be converted into a valid sequence of LKMM ppo and rfe

relations. This also conveniently encodes the rmw-sequence relation used in the LKMM's prop relation.

Compound prop. Finally, we modify the propagation relation to incorporate rmw-sequence through st rather than handling it separately:

$$\text{prop} = (\text{overwrite} \cap \text{ext})^? ; \text{cumul-fence} ; \text{st}^?$$

For the cumulative fences that accept a preceding rfe (po-rel and strong-fence), the cumul-fence relation uses st in that position instead. This subsumes both the rfe preceding the fence and the rmw-sequence following the previous cumulative fence, since st admits sequences of rf;rmw. The terminal rfe of prop is similarly replaced by st. This reorganization does not change which events prop can relate in a pure Linux program.

The LKMM's propagates-before axiom is unchanged in form—acyclic prop;strong-fence;hb*—now read with the compound prop and hb.

4.2 Generalized Composition

One key challenge in defining a compound model is allowing C's synchronization patterns while respecting Linux's fence requirements. In order to do so, we redefine relation composition to allow for relations to relate between events and relations rather than only between events. The intuition behind this change in definition is that rather than allowing a synchronization to end in a fence, as C defines synchronization, we allow it to end in the fence plus its entire context. This enables *fence context sharing* seen in Figure 5.

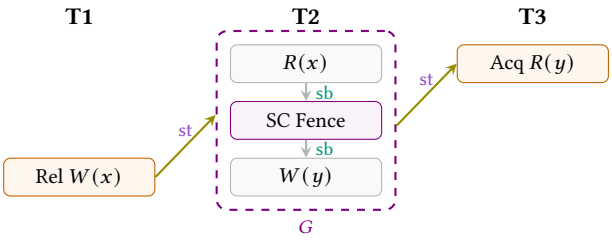

Figure 5. Fence context sharing. The dashed box marks the fence context $G = (R(x), \text{SC Fence}, W(y))$.

Both release and acquire synchronize with the fence and its context, similar to how C would allow synchronization with the fence directly, while Linux fences remain inside their necessary contexts.

To formalize this, we consider a generalization of relations. These are defined as sets of sequences $(s_1, s_2, \ldots, s_n)$ where the s_i can be events or, inductively, other sequences. Relations thus become the special case $\subseteq \{(e_1, e_2) \mid e_1, e_2 \in \text{Events}\}$.

For these generalized relations, we also define a different notion of composition. Let $\text{first}((s_1, \ldots, s_n)) = s_1$ and $\text{last}((s_1, \ldots, s_n)) = s_n$ return the first or last element of the sequence; an element may itself be an event or a nested sequence. We write $\text{last}^m(s)$ and $\text{first}^m(s)$ for m repeated applications of last and first, respectively; e.g., $\text{last}^2([A, [B, C]]) = \text{last}([B, C]) = C$. We say two sequences compose when the first or last sequences agree at some level, more formally:

$$\text{composes}(s_1, s_2) \text{ iff } \exists m, n \geq 0, m > 0 \vee n > 0, \\ \text{last}^m(s_1) = \text{first}^n(s_2) \qquad (1)$$

We then define the composition of generalized relations (sets of sequences) as:

$$r_1 ; r_2 := \{(s_1, s_2) \in r_1 \times r_2 \mid \text{composes}(s_1, s_2)\} \qquad (2)$$

The generalized version allows overlap on intermediate sub-relations—fence contexts—while requiring at least one application of first or last to prevent trivial self-composition.

This mechanism preserves the LKMM's requirement that fences remain in context: the fence never appears as a standalone endpoint. Pattern-dependent fences like `Before_atomic` are only credited with ordering when their required context (e.g., a following RMW) is part of the composed unit. At the same time, the entire context can participate in multiple synchronizations, recovering C's fence patterns.

The trivial filter. Interpreting the LKMM with generalized relations leads to a subtle yet important distinction when generalizing the identity relation, id:

$$\text{id} = \{s \mid \exists e \in \text{Event}, \text{composes}(s, (e)) \\ \wedge \text{composes}((e), s)\} \qquad (3)$$

To understand the issue, consider the following two variants of the message passing litmus, differing only in an acquire being replaced by a read followed by an acquire fence. Both variants should be equivalent, and both behaviors forbidden:

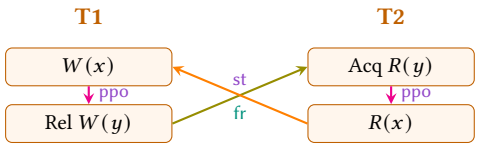

Figure 6. Message passing with an acquire

Variant 1: with an acquire (Figure 6). The prop relation relates $R(x) \xrightarrow{\text{fr}} W(x) \xrightarrow{\text{ppo}} \text{Rel } W(y) \xrightarrow{\text{st}} \text{Acq } R(y)$, returning to T2. Combined with ppo from Acq $R(y)$ to $R(x)$, this forms a cycle. The prop relation relates $R(x)$ to Acq $R(y)$—these are different events and thus are related by $(\text{prop} \setminus \text{id}) \cap \text{int}$ so the stale read is forbidden.

Variant 2: with an acquire fence (Figure 7). The prop relation relates $R(x) \xrightarrow{\text{fr}} W(x) \xrightarrow{\text{ppo}} \text{Rel } W(y) \xrightarrow{\text{st}} R(x)$, since the st relation can no longer end at the read of y that has been demoted from an acquire. The prop relation starts and ends on the same event: it is an identity relation. Under the LKMM's

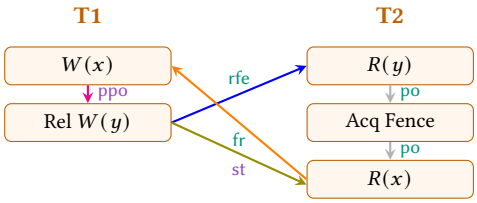

Figure 7. Message passing with an acquire fence

prop \ id, this relation would be filtered out, incorrectly permitting the stale read.

To deal with this, we replace the id in prop \ id with trivial.

$$\begin{aligned}\text{trivial} = \{&(e), (e,e), (e,(e,e)), ((e,e),e),\\ &(e,e,e), \ldots \mid e \in \text{Event}\}\end{aligned} \tag{4}$$

A sequence is trivial if every event in it—at every level of nesting—is the same event. This is strictly stronger than id: the prop edge in Variant 2 traverses fr, ppo, and st, so it starts and ends on the same event but is not trivial. The cycle is preserved and the stale read is correctly forbidden. This change does not affect pure Linux programs.

4.3 Full Model Definition

Having explained the extensions to support C, we are ready to define the compound C-LKMM memory model. The compound model maintains each axiom of the LKMM and adds the no-thin-air axiom specific to C:

Axiom	Definition
Per-location SC	acyclic $(\text{po}_{=\text{loc}} \cup \text{com})$
Atomic RMW	empty $(\text{rmw} \cap (\text{fre}\fatsemi\text{coe}))$
Happens-before	acyclic hb
Propagates-before	acyclic pb
No-thin-air	acyclic $[\text{Plain}]\fatsemi(\text{po} \cup \text{rf})$

where:

$$\begin{aligned}\text{acq-po} = {} &\{(a,b) \mid a.str \geq \text{acq}\}\\ &\cup \{(a,f,b) \mid f{=}\text{Fence},\ f.str \geq \text{acq}\}\\ &\cup \{(a,f,g,b) \mid f{=}\text{Fence},\ f.str \geq \text{acq},\ g{=}\text{Fence}\}\end{aligned} \tag{5}$$

$$\begin{aligned}\text{po-rel} = {} &\{(a,b) \mid b.str \geq \text{rel}\}\\ &\cup \{(a,f,b) \mid f{=}\text{Fence},\ f.str \geq \text{rel}\}\\ &\cup \{(a,f,g,b) \mid g{=}\text{Fence},\ g.str \geq \text{rel},\ f{=}\text{Fence}\}\end{aligned} \tag{6}$$

$$\begin{aligned}\text{inter-start} = {} &[\text{Marked} \cup \text{rel}] \cup \text{po-rel}\\ &\cup\ \text{strong-fence} \cup \text{weak-fence}\end{aligned} \tag{7}$$

$$\begin{aligned}\text{inter-end} = {} &[\text{Marked} \cup \text{acq}] \cup \text{acq-po}\\ &\cup\ \text{strong-fence} \cup \text{weak-fence}\end{aligned} \tag{8}$$

$$\text{st} = \text{inter-start}\fatsemi(\text{rfe} \cup (\text{rf}\fatsemi\text{rmw})^+\fatsemi\text{rf}^?)\fatsemi\text{inter-end} \tag{9}$$

$$\text{hb} = \text{ppo} \cup \text{st} \cup (\text{prop} \setminus \text{trivial}) \cap \text{int} \tag{10}$$

$$\text{pb} = \text{prop}\fatsemi\text{strong-fence}\fatsemi\text{hb}^* \tag{11}$$

$$\text{prop} = (\text{overwrite} \cap \text{ext})^?\fatsemi\text{cumul-fence}\fatsemi\text{st}^? \tag{12}$$

The relations compose using generalized composition (Equation 2), indicated by the symbol ⨾. The RC11 no-thin-air axiom is adapted to apply only when every access in the cycle is Plain—in our setting, a C access (Table 1). This is the weakest guarantee needed to match C's behavior and avoids imposing unnecessary restrictions on orderings between Linux operations in compound programs. Restricting the event set to a single language collapses the compound model onto that language's own: st reduces as described in §4.1, and Theorems 1 and 2 (§5) show that the remaining relations likewise recover the corresponding base-model behaviors. The machinery the compound model adds—mixed-language synchronization and fence contexts—is thus confined to the boundary; much of its formal complexity is inherited from RC11 and the LKMM, not introduced by the composition.

5 Soundness

We prove three properties of the compound model: it preserves the behavior of pure C programs (Theorem 1), it preserves the behavior of pure Linux programs (Theorem 2), and the shared synchronization primitives are interchangeable across the boundary (Theorem 3). We provide proof sketches here; full proofs are in the extended version [13].

5.1 C Preservation

> ▸ **Theorem 1** (C Preservation). On a program consisting entirely of C operations (excluding SC atomics), an execution is allowed by the compound model if and only if it is allowed by RC11.

The proof proceeds in two directions.

Forward: compound disallows ⇒ C disallows. We prove any cycle forbidden by the compound model is also forbidden by C. The argument has three parts, one for each axiom.

Happens-before. A cycle in compound hb^+ must map to a violation of C's coherence axiom, $\text{acyclic}(\text{hb}_C\fatsemi\text{eco}^?)$. The key difficulty is that compound hb uses prop, which can introduce multiple non-st communication relations in a single cycle, whereas C's coherence axiom permits at most one. Lemma 5.1 resolves this.

Lemma 5.1 (Prop reduction). Let σ be a cycle in compound hb^+ containing $k \geq 2$ relations from $(\text{prop} \setminus \text{trivial}) \cap \text{int}$. Then either:

(a) σ implies a violation of per-location SC ($\text{acyclic}(\text{po}_{=\text{loc}} \cup \text{com})$), or

(b) there exists cycle σ' in hb^+ with ≤ 1 prop relation.

With Lemma 5.1 in hand, any compound hb cycle with at most one prop relation maps directly to C's coherence axiom. The compound relations translate as follows: $\text{ppo} \subseteq \text{sb}$ (since all of sb is preserved in C), $\text{st} \subseteq (\text{sb} \cup \text{sw})^*$ (since every st between C operations requires release-acquire, which gives sw), and a single prop relation maps to the $\text{eco}^?$ suffix in C's axiom. Composing the above observations gives us that a

cycle in compound hb^+ is a cycle in $(\mathsf{sb} \cup \mathsf{sw})^+ ; \mathsf{eco}^?$, which is exactly what C's coherence axiom forbids.

Propagates-before. A cycle in compound pb^+ must imply a cycle in C's pscf. Rewriting pscf as:

$$[\mathtt{F} \cap \mathtt{SC}] ; \mathsf{hb} ; (\mathsf{eco} ; \mathsf{hb})^? ; [\mathtt{F} \cap \mathtt{SC}]$$

Then expanding a cycle constructed from these relations:

$$\ldots ; (\mathsf{eco} ; \mathsf{hb})^? ; [\mathtt{F} \cap \mathtt{SC}] ; \mathsf{hb} ; (\mathsf{eco} ; \mathsf{hb})^? ; [\mathtt{F} \cap \mathtt{SC}] ; \mathsf{hb} ; \ldots$$

We can rewrite the definition of pscf to start and end at a different point in the cycle rather than at the fences:

$$(\mathsf{eco} ; \mathsf{hb})^? ; [\mathtt{F} \cap \mathtt{SC}] ; \mathsf{hb}$$

Of which $\mathsf{prop};\mathsf{strong\text{-}fence};\mathsf{hb}^*$ is a subset since the strong-fence relation implies at least one $\mathsf{po} \subset \mathsf{hb}$ relation before and after the fence.

Per-location SC. A cycle in $\mathrm{acyclic}(\mathsf{po}_{=\mathrm{loc}} \cup \mathsf{com})$ maps directly to C's coherence axiom: $\mathsf{po}_{=\mathrm{loc}} \subseteq \mathsf{sb}$, so the cycle is in $(\mathsf{sb} \cup \mathsf{sw})^+ ; \mathsf{eco}^?$ with sw vacuous. If the cycle is entirely in com (no $\mathsf{po}_{=\mathrm{loc}}$), then by replacing $\mathsf{rf};\mathsf{fr}$ with co, every write is coherence-ordered before itself, contradicting the definitional irreflexivity of co.

Reverse: C disallows ⇒ compound disallows.

Coherence. A cycle in $(\mathsf{sb} \cup \mathsf{sw})^+ ; \mathsf{eco}^?$ either contains sw or it does not. If it does not, then this reduces to a cycle in per-loc-SC: every operation in the simplified cycle is on the same location so replace the sb relations with $\mathsf{po}_{=\mathrm{loc}}$. Otherwise for cycles that do contain sw, if the cycle does not contain an eco relation then every thread communicates using sw and thus every po relation can be replaced with po-rel or strong-fence as every thread will have a release or stronger as part of its synchronization with other threads. sw relations map to st and thus the cycle is in compound hb^+. If it does have an eco relation, then this cycle is disallowed via prop: the eco relation is the start of the prop relation, then the cumulative fences of prop are a superset of $(\mathsf{sb} \cup \mathsf{sw})^+$.

PSCF. A cycle in pscf implies a cycle in pb by the same rotation argument: $(\mathsf{eco};\mathsf{hb}_C)^? \subseteq \mathsf{prop}$ and compound hb subsumes C's hb, so $\mathsf{prop};\mathsf{strong\text{-}fence};\mathsf{hb}^*$ captures the rotated cycle.

5.2 LKMM Preservation

> ▶ **Theorem 2** (LKMM Preservation). On a program consisting entirely of Linux operations, an execution is allowed by the compound model if and only if it is allowed by the LKMM.

Proof sketch. The compound model is defined as an extension of the LKMM, so both directions reduce to subset arguments.
Forward: compound disallows ⇒ LKMM disallows. When restricted to Linux operations, ppo and st reduce to Linux ppo and rfe, and prop either reduces to Linux prop or a per-location SC cycle, and strong-fence is unchanged. Therefore any cycle in compound hb is a cycle in LKMM hb, and any cycle in compound pb is a cycle in LKMM pb.
Reverse: LKMM disallows ⇒ compound disallows. The compound ppo, st, prop, and strong-fence are supersets of their Linux equivalents. Therefore any LKMM hb cycle is a compound hb cycle and any LKMM pb cycle is a compound pb cycle. □

5.3 Interchangeability

The natural formulation of interchangeability—replacing a Linux acquire with a C acquire preserves all allowed behaviors—is too strong. Consider an address dependency between two Linux reads:

$$\text{T1:} \quad r_0 = R(x); \quad \text{Acq } R(*r_0)$$

The LKMM preserves the address dependency as ppo regardless of the acquire annotation. If the Linux acquire is replaced by a C acquire, the address dependency is no longer preserved: C does not recognize address dependencies. The behavioral change is not a failure of the acquire primitive—it is a consequence of C's different treatment of intra-thread ordering.

We therefore define interchangeability as preservation of the primitive's own guarantees: its synchronization capability and its ordering contribution.

> ▶ **Theorem 3** (Interchangeability). For each shared primitive $p \in \{\text{release, acquire, SC fence}\}$:
>
> (a) *Synchronization:* st does not distinguish between a C instance and a Linux instance of p. A C release synchronizes with any acquire (C or Linux), and vice versa.
>
> (b) *Ordering:* The ordering relations po-rel, acq-po, and strong-fence do not distinguish between a C instance and a Linux instance of p. A C acquire orders subsequent operations via acq-po identically to a Linux acquire.

Proof sketch. Interchangeability follows directly from the construction of the model: the ordering guarantees of an acquire are stated by the acq-po relation, which does not differentiate between Linux and C acquires. The synchronization guarantees of an acquire are defined in st, which does not distinguish between Linux and C acquires. The same is true for releases and SC fences, using po-rel and strong-fence relations for ordering guarantees and st for synchronization guarantees. □

5.4 Model Implementation and Litmus Testing

Implementation. We implemented a Python tool that encodes the compound model and serves as a litmus test engine. Existing tools such as `herd7` [3] and Dartagnan [22] cannot express the generalized fence composition introduced in §4, as it requires chaining more than two events. Our

engine enumerates all candidate executions, evaluates the compound model's axioms, and reports the set of allowed outcomes. All litmus results below are produced by this tool, which will be released as part of the artifact.

Litmus testing results. We drew base tests from two sources: the C suite from Dartagnan [22], covering Message Passing, Load Buffering, Store Buffering, IRIW, and ISA2; and the LKMM suite distributed with the Linux kernel source tree, including WRC and Coherence, and systematically expanded each base test by varying the strengths of its operations. Across all 3,272 resulting tests, the allowed outcomes under the compound model were identical to those of the corresponding single-language model, supporting the C-preservation and LKMM-preservation theorems (§5).

To validate interchangeability, we constructed compound litmus tests by systematically replacing C synchronization primitives with their LKMM counterparts and vice versa: release stores, acquire loads, and fences. The interchangeability tests specifically exercise the two novel mechanisms of the compound model: *st* and fence-context sharing. These substitutions, especially of fences, require the synchronization to survive while retaining the LKMM fence context, so their agreement provides an independent executable check of the generalized composition machinery in addition to the formal proofs. All 11,584 resulting tests matched the outcomes of the original, supporting the interchangeability theorem (§5.3). Table 5 summarizes the results.

Suite	# of Tests	Outcomes
C (single-language)	1868	✓
Linux (single-language)	1404	✓
Compound	11584	✓

Table 5. Litmus test validation summary. ✓ = in every test in the suite, the set of allowed outcomes under the compound model is identical to that of the reference model (single-language for C/LKMM rows; original test for compound row).

Thus the C and LKMM corpora cross-check Theorems 1 and 2, respectively, while the 11,584 systematically substituted tests cross-check Theorem 3 using the executable compound model. Mechanizing the three theorems (e.g., in Lean) is worthwhile future work; for now, the executable cross-check guards against the case-analysis errors to which hand proofs of such models are susceptible [15]. These 14,856 tests deliver three things: a precise, executable specification of the compound model serving as a reference implementation; empirical support for all three theorems; and a corpus of 11,584 compound litmus tests that gives future cross-boundary verification tools a concrete test target.

6 Case Studies

We demonstrate the compound model's utility both *constructively*—as a design guide for lock-free data structures shared between user-space (C) and kernel-space (LKMM) via eBPF

```
while (retries) {
  tail = load_acq(queue->tail); // C
  tail = load_rlx(queue->tail); // LKMM
  next = load_acq(tail->next);
  if (next != NULL) {
    cas_rel(queue->tail, tail, next);
      continue;
  }
  if (cas_rel(tail->next, next, new_node) == next)
    break;
  continue;
}
if (cas_rel(queue->tail, tail, new_node) != tail)
  return SUCCESS
return SUCCESS
```

Figure 8. Sync. step after payload write. P_C (orange) and P_L (blue) for MSQ

(§6.1)—and *diagnostically*—as a reasoning tool that can expose missing ordering and confirm existing ordering already suffices (§6.2).

6.1 Constructive: eBPF Lock-Free Data Structures

BPF arenas provide regions of memory that both userspace and the kernel can directly access (using real pointers), without a syscall, which would add high overhead. They create a platform for developing shared data structures that span the user-kernel boundary, but they leave open questions about safe synchronization. The formalization of a BPF memory model is ongoing work [6, 18]; we treat eBPF kernel code as LKMM-governed, which is consistent with eBPF's proposed model for all operations our case studies use. Our analysis assumes that the BPF toolchain preserves the LKMM-relevant ordering used here: synchronization operations remain explicit through bytecode and JIT lowering—BPF atomics and fences are explicit instructions—and the verifier is a static safety checker that does not reorder memory operations. Dependency ordering carries the same guideline-conditioned assumption as source-level LKMM code (§2). We do not attempt to formalize the BPF compiler, verifier, or JIT themselves.

We studied five data structure implementations from widely used concurrent data structure libraries: the Michael-Scott Queue from the Concurrent Data Structures (CDS) library, Facebook Folly's SPSC Queue, ConcurrencyKit's FIFO Queue, Ring, and Stack. Together, they exercise the producer-consumer synchronization patterns that recur throughout user-kernel concurrent code. We present the Michael-Scott Queue in detail below; the remaining four follow the same process and are presented in the extended version [13].

For each data structure, we began our work with the reference C/C++ implementation, and apply the **DARE** process:

1. **Distill**: From the reference C implementation, we extract the memory-model agnostic algorithms of the producer

```
while (retries) {
  head = load_acq(queue->head); // C
  head = load_rlx(queue->head); // LKMM
  next = load_acq(head->next);
  if (load_acq(queue->head) != head)
    continue;
  if (next == NULL)
    return FAIL;
  tail = load_acq(queue->tail);
  if (head == tail) {
    cas_rel(queue->tail, tail, next);
    continue;
  }
  read_data = next->data;
  if (cas_acq(queue->head, head, next) == head)
    return SUCCESS
  continue;
}
return FAIL
```

Figure 9. Sync. step before payload read. C_C (orange) and C_L (blue) for MSQ

(P) and consumer path (C). We use the data structures' invariants to identify the algorithms' required orderings.
2. **Annotate for C**: We re-express P and C as P_C and C_C respectively, annotating each shared-memory access with the C primitive (`relaxed`, `acquire`, `release`) needed to enforce the required orderings under C's `release-acquire`-based happens before.
3. **Relax for LKMM**: We re-express the same algorithms as P_L and C_L. Wherever the LKMM's preserved program order (ppo) already provides a required ordering — through an address dependency, a data dependency, a control-plus-write, or a fence — we drop the corresponding C annotation. The result is a Linux-side implementation in which the programmer makes explicit only the orderings that ppo does not already cover.
4. **Evaluate Heterogeneous Synchronization**: We use the compound model to reason about cross-domain pairings: a kernel-producer paired with a user-consumer, or vice-versa. The compound model's interchangeability property tells us that the shared synchronization primitives carry the same meaning on both sides and lets us reason about the synchronization.

Identifying the invariants an implementation must preserve and distilling them into critical litmus tests (the Distill step) remain manual—as they would under any memory model, since they require understanding the data structure itself. What the executable model automates is the Evaluate step: deciding whether an invariant-violating outcome is allowed.

6.1.1 The Michael-Scott Queue (MSQ).

MSQ [20] is a simple non-blocking queue that uses linked lists. The root node has `head` and `tail` pointers, which point to the head and tail nodes respectively. Each node contains a payload. P operations insert nodes at the `tail` (Figure 8), and C operations delete nodes from the `head` (Figure 9). Throughout the following case studies, `load_rlx` denotes `READ_ONCE` for Linux and a C relaxed atomic load (`memory_order_relaxed`) for C. Similarly, `load_acq` denotes `smp_load_acquire` for Linux and a C acquire atomic load (`memory_order_acquire`) for C, and `store_rel` denotes `smp_store_release` for Linux and a C release atomic store (`memory_order_release`) for C. A `cas_X` is a compare-and-swap with `X` the success order and `relaxed` the failure order—the kernel's `cmpxchg_X` and C's `compare_exchange`, respectively.

Invariants.

1. If the `head` and `tail` point to the same address (node), the queue is empty. At initialization, the `head` and `tail` point to a dummy node.
2. Insertions always take place at the `tail`. The tail node's next pointer always points to `NULL`. If the tail node's next pointer appears to be non-`NULL`, it means there was a racing insertion (the `tail` pointer is lagging behind).
3. Racing insertions are easy to detect (`tail->node.next != NULL`). If these are detected, the insertion algorithm attempts to adjust the tail pointer to the correct `tail` node and only then performs the insertion (after asserting `tail->node.next == NULL`).
4. Deletions advance the `head` pointer to skip the current head node, re-pointing it to the next head node.

Required Orderings. *Invariant 1* is structural—it defines the empty-queue configuration—and carries no ordering requirement. *Invariant 2* requires that the new node's `next` pointer be visible (as `NULL`) to any thread that observes it as the new `tail`. *Invariant 3* requires that a consumer that observes the new `tail->next` also observes the corresponding payload write. *Invariant 4* requires that `head` advances only after the new head node is fully linked.

P_C: The C Producer. Figure 8 shows the C-annotated producer, with each annotation mapping to one of the orderings above. The algorithm requires `load_acq(queue->tail)` so the subsequent dereference `tail->next` observes a fully initialized node *(Invariant 2)*. `load_acq(tail->next)` is required since the value read drives a branch and CAS *(Invariant 3)*; without it, C does not order the load against either. The CAS `releases` on `tail->next` and `queue->tail` are required so that the new node's contents — in particular its `NULL` next pointer and its payload — are visible to any future thread that reads the updated `tail` or `tail->next`.

P_L: The LKMM Producer. Figure 8 also shows the LKMM-annotated producer. The C annotation on `load_acq` (queue->tail) becomes `READ_ONCE`: the LKMM's addr-dependency ppo from `tail` to `tail->next` preserves the ordering of the dereference, precisely what *Invariant 2* needs. The CAS `releases` remain, since the LKMM does not provide release

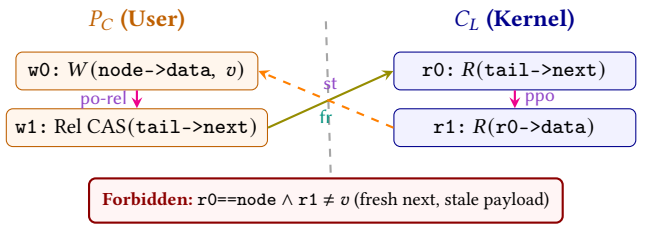

Figure 10. Heterogeneous pairing: user producer (P_C) with kernel consumer (C_L) for the Michael-Scott queue. The st relation bridges the boundary from the user's `cas_rel` to the kernel's `ld_rlx`. $\texttt{w0} \xrightarrow{\text{po-rel}} \texttt{w1} \xrightarrow{\text{st}} \texttt{r0} \xrightarrow{\text{ppo}} \texttt{r1} \xrightarrow{\text{fr}} \texttt{w0}$ is forbidden by the compound model. The algorithm produces at the `tail` and consumes from the `head`; the two meet at the first append. When the producer links a node into the empty queue, `head==tail` (the dummy node), so the producer's write of `tail->next` and the consumer's read of `head->next` access the same location, and the figure shows exactly this pairing.

semantics for free; they are written as `cas_rel`. The producer thus drops one `acquire` relative to the C version, recovering them through address and control dependencies.

The consumer follows a similar pattern: one acquire is dropped on the LKMM side (`load_acq(queue->head)`) via an addr dependency, and the CAS `release` is preserved.

Evaluate Heterogeneous Pairing P_C with C_L. The most interesting cross-language pairing for MSQ is the user producer paired with the kernel consumer: a user process enqueues events, and a kernel component consumes them. We extract the cross-boundary litmus test and use the compound model to forbid the outcome that would violate *Invariant 3*.

Invariant 3 requires that a consumer observing the new `tail->next` also observes the corresponding payload. The violating outcome is `r0 = node` $\wedge$ `r1` $\neq v$ in Figure 10—a fresh `next` pointer with a stale payload. The compound model rules it out by a single happens-before chain. On the C producer side P_C in Figure 10, there is a `po-rel` edge from the payload write (`w0`) to the `release` (`w1`). The `st` edge bridges the boundary: the producer's `cas_rel` (`w1`) satisfies `inter-start` (C release), and the consumer's plain read (`r0`) satisfies `inter-end` (LKMM, no acquire needed). An address-dependency `ppo` edge from `r0` to `r1` then carries the ordering forward, forbidding the stale read and preserving Invariant 3.

The reverse pairing of P_L with C_C is symmetric: the kernel-side `cas_rel` satisfies `inter-start` trivially (LKMM, no release annotation needed), and the C consumer's `load_acq` satisfies `inter-end`, yielding the same happens-before edge via `st`.

Result. All four pairings of MSQ preserve the queue's invariants under the compound model. The matched `release` / `acquire` pattern that closes the KCOV bug from Figure 1 appears here in generalized form: in every cross-boundary pairing, the C side contributes its `release` or `acquire` annotation, the LKMM side contributes ppo, and the compound model's `st` relation joins them across the boundary. The remaining four data structures (Folly SPSC, CK FIFO, CK Ring, CK Stack) follow the same DARE process and are presented in the extended version [13]; Table 6 summarizes all five.

6.1.2 Performance.

To measure performance for both (P_L, C_C) and (P_C, C_L) pairings, our infrastructure allocates two buffers, which we name KU-buffer (for kernel→user communication), and the UK-buffer (for user→kernel communication). For each queue entry, the flow is as follows:

1. **T0**: external syscall (`inode_create`) triggers P_L on KU, producing a single queue entry.
2. **T1**: user-thread polls on KU (C_C) and relays the queue entry to UK (P_C).
3. **T2**: kernel thread consumes the entry from UK (C_L).

We measure the total end-to-end latency for each queue item to ping-pong from kernel mode to user mode and back: $T_{E2E} = T_{P_L} + T_{C_C} + T_{P_C} + T_{C_L}$.

We use an adapted `io_uring`/liburing path as a general-purpose reference point rather than as a directly matched specialized baseline. Our goal is to evaluate whether the compound-model-guided implementations preserve correctness without sacrificing practical performance. On this workload, the CK FIFO, Folly SPSC, and CK Ring implementations are 8–12% faster than the adapted `io_uring` path, while the remaining structures remain within the same practical performance regime. All experiments were run on the hardware and software configuration described in Table 7; results are presented in Table 6.

These results show that the synchronization required by the compound model does not impose prohibitive overhead: the resulting cross-boundary data structures remain competitive with a mature general-purpose kernel interface, and several are faster on the measured producer-consumer workload.

6.2 Diagnostic: Cross-Boundary Analysis

The compound model is also useful for reasoning about existing cross-boundary code. We have already seen one example: the `kcov_remote` bug (§1, Figure 1), where the model exposed a missing write barrier on the kernel side and a missing acquire on the user side, both of which were required to close the happens-before chain. We now apply the same reasoning to a second case where the conclusion is the opposite: the existing code is already correct.

6.2.1 The io_uring Completion Ring: A Non-Bug.

The io_uring completion queue shares a ring buffer between kernel and user space. The kernel writes CQEs and publishes them by advancing a tail index with a release store. The user-space consumer reads the ring as follows:

DS	P_C	P_L	C_C	C_L	ppo	E2E lat.	Mops/s
Folly SPSC	1 acq 1 rel	1 rel	1 acq 1 rel	1 acq 1 rel	ctrl	**330 ns**	**3.04**
CK Ring SPSC	1 acq 1 rel	1 rel	1 acq 1 rel 1 mb	1 acq 1 rel	acq	**334 ns**	**3.00**
CK FIFO SPSC	1 rel	1 rel	1 acq 1 rel	-	addr	**339 ns**	**2.96**
CK Stack UPMC	1 rel-CAS	1 rel-CAS	1 acq 1 acq-CAS	1 rlx-CAS	addr and data	565 ns	1.77
MSQueue	2 acq 3 rel-CAS	1 acq 3 rel-CAS	4 acq 1 rel-CAS 1 acq-CAS	3 acq 1 rel-CAS 1 acq-CAS	addr and ctrl	926 ns	1.08
io_uring/liburing	-	-	-	-		369 ns	2.71

Table 6. Synchronization annotations across the four implementations of each data structure, along with end-to-end latency and throughput (averaged from 1000 runs).

Component	Specification
Processor	Intel Core i5-7600 @ 3.50 GHz (Max 4.10 GHz)
Topology	4 Physical Cores (No SMT), Single NUMA Node
Caches	128 KiB L1, 1 MiB L2, 6 MiB L3
Memory	16 GB DDR4-2400 (Single-Channel)
OS Kernel	Linux v6.18.2
Compiler	Clang/LLVM 20.1.8
BPF Toolchain	bpftool v7.5.0, libbpf v1.5

Table 7. Hardware and software configuration

```
r0:  tail = smp_load_acquire(ring->cq.ktail);
r1:  head = *ring->cq.khead;
     if (head != tail)
r2:    cqe = ring->cq.cqes[(head & mask)];
```

There is an address dependency from `r1` to `r2`, and C does not enforce ordering for address dependencies. We initially submitted a patch promoting `r1` to `smp_load_acquire()`; the patch was accepted upstream. On closer analysis using the compound model, we determined that the original code was already correct (Figure 11).

The critical observation is that `khead` is written only by the user thread to mark consumed entries; the kernel reads it only for free-space accounting. On the consumer path, the user's load of `khead` thus reads its own prior write, so the address dependency from `r1` to `r2` is off the synchronization critical path: the cross-boundary synchronization runs entirely through `ktail`, whose kernel-side release pairs with the user-side acquire via st (Figure 11). The acquire on `khead` was unnecessary.

What does this example reveal—is the compound model hard to reason about? The question our patch answered—must this address dependency be backed by an acquire?—is fundamentally a C question. The subtlety is C's: the question arises only because C declines to order address dependencies (§2), and its answer here is that no acquire is needed, because the load reads the user's own prior write. What the compound model adds is the compositional structure: each side's obligations are judged under its own model, and the composition works as expected.

This case complements `kcov`. In `kcov`, *both* loads of the dependency chain read kernel-written data—the dependency *is* the critical path—and both fixes were needed (§1). In io_uring, `r1` reads the user's own state, and the existing `ktail` acquire suffices. The compound model's value lies in both directions: diagnosing missing ordering, *and* certifying existing code correct.

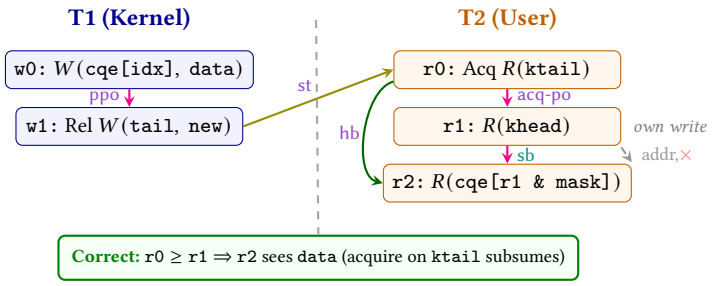

Figure 11. The io_uring completion queue under the compound model. The kernel writes a CQE and releases the tail (`w0`–`w1`). The user acquires the tail (`r0`), reads the head (`r1`, its own bookkeeping), and dereferences (`r2`). The hb chain $\texttt{w0} \xrightarrow{\text{ppo}} \texttt{w1} \xrightarrow{\text{st}} \texttt{r0} \xrightarrow{\text{hb}} \texttt{r2}$ guarantees `r2` sees the kernel's write. The address dependency from `r1` to `r2` (greyed, ×) is formally redundant.

7 Related Work

Language-Level Memory Models. The Java Memory Model was the first language-level memory model specification [16]. The C/C++11 standard followed with a richer design featuring relaxed, release-acquire, and sequentially consistent atomics; Batty et al. gave the first formal semantics [4], and Lahav et al. repaired the SC axiom to produce RC11 [15], which forms the basis of the current C standard [12]. These efforts target a single language in isolation; our work addresses what happens when two such models must coexist on shared memory.

The Linux Kernel Memory Model. Howells and McKenney's informal treatment of kernel memory ordering [11, 17] laid the groundwork for the LKMM, which was formalized by Alglave et al. as an axiomatic model in the `cat` language [2]. McKenney et al. have also considered how LKMM primitives map to C atomics [19], observing that many kernel accessors have C11 counterparts. Our work is the first to formalize this interface. The Rust-for-Linux effort defines Rust

atomics to be deliberately LKMM-faithful—`Relaxed`, for instance, carries address/data/control dependencies, unlike C `Relaxed` [7]. Our compound model's LKMM basis is therefore consistent with Rust-for-Linux semantics, suggesting applicability to Rust-kernel ↔ user-space C boundaries.

Compound Memory Models. The concept of compound memory models—formal composition of distinct consistency specifications over shared memory—was introduced in the Primer [21]. Goens et al. gave the first full formalization for composing hardware-level ISA memory models [9]. These efforts operate at the hardware level; in contrast, our work is the first to compose a language-level model (C) with an OS-level model (LKMM)—two specifications with structurally incompatible notions of happens-before that coexist at the software level.

Formal Tools. Alglave et al.'s `herd7` tool and the `cat` language [3] enable axiomatic memory model specification and litmus test evaluation. For verification of concurrent C programs, Dartagnan [8, 22] performs bounded model checking against axiomatic models, and GenMC [14] and Nidhugg [1] provide stateless model checking for C/C++ concurrency. These tools currently target single-model programs; adapting them to check cross-boundary programs against the compound model is a direction for future work.

8 Conclusion

We presented the C/LKMM compound memory model, which composes the C and Linux kernel memory models into a single compound model that preserves each model's semantics in isolation and bridges them at shared-memory boundaries. The model has both diagnostic and constructive value. Applied to existing kernel code, it exposed a real concurrency bug in `kcov` (fix accepted upstream). Applied to new designs, it guided the construction of five cross-boundary lock-free data structures over eBPF.

As eBPF, io_uring, and similar zero-copy interfaces continue to blur the boundary between user space and the kernel, the need for formal cross-boundary reasoning will only grow. Our experience suggests that composability—the ability to bridge two models at a shared boundary—warrants consideration as a design criterion alongside correctness and performance in future memory model work. Our compound model provides the foundation for that reasoning.

Acknowledgments

We thank Paul McKenney, Alan Stern, and the anonymous shepherd for constructive feedback on this work. This work is supported in part by the National Science Foundation under grants CCF-2525270 and CNS-2245999, and by ARM. Generative AI tools (including ChatGPT Codex, Claude, and GitHub Copilot) were used to assist the development of the Compound Model litmus testing engine (§5.4) and the eBPF heterogeneous data structure framework (§6). AI tools were also used to code the diagrams in this paper and copyedit the text.

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–367.

[2] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. ACM, 405–418. doi:10.1145/3173162.3177156

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats. *ACM TOPLAS* 36, 2 (jul 2014), 1–74. doi:10.1145/2627752

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. doi:10.1145/1926385.1926394

[5] Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiadis. 2018. P0982R1: Weaken Release Sequences. ISO/IEC JTC1/SC22/WG21 paper, https://wg21.link/p0982.

[6] Jonathan Corbet. 2024. A memory model for BPF. LWN.net. https://lwn.net/Articles/976071/.

[7] Boqun Feng. 2026. [PATCH 0/2] Provide Rust atomic helpers over raw pointers. https://lore.kernel.org/all/20260120115207.55318-1-boqun.feng@gmail.com/

[8] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 355–373. doi:10.1007/978-3-030-25540-4_19

[9] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound memory models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1145–1168.

[10] Paul Heidekrüger and M Elver. 2022. Status Report: Broken Dependency Orderings in the Linux Kernel. Linux Plumbers Conference.

[11] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. 2024. memory-barriers.txt. Linux kernel documentation, https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[12] International Organization for Standardization 2024. *ISO/IEC 9899:2024 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland.

[13] Hastin Kapoor, Soham Bagchi, An Qi Zhang, Soham Chakraborty, Marco Elver, Andrés Goens, Susmit Sarkar, Ryan Stutsman, and Vijay Nagarajan. 2026. Formalizing the User-Kernel Shared-Memory Boundary: The C/LKMM Compound Model (Extended Version). Zenodo preprint. doi:10.5281/zenodo.22970154

[14] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 96–110. doi:10.1145/3314221.3314609

[15] 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, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. doi:10.1145/3062341.3062352

[16] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 378–391. doi:10.1145/1040305.1040336

[17] Paul E. McKenney. 2017. *A formal kernel memory-ordering model (part 1).* https://lwn.net/Articles/718628/

[18] Paul E. McKenney and Puranjay Mohan. 2024. Towards a BPF Memory Model. Linux Plumbers Conference. https://lpc.events/event/18/contributions/1949/.

[19] Paul E. McKenney, Ulrich Weigand, Andrea Parri, and Boqun Feng. 2016. Linux-Kernel Memory Model. http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0124r2.html. ISO/IEC JTC1/SC22/WG21 P0124R2.

[20] Maged M Michael and Michael L Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. 267–275.

[21] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence* (2 ed.). Morgan & Claypool Publishers.

[22] Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 378–382.

Artifact Appendix

This paper is accompanied by two artifacts (Appendix A and B). The first one reproduces the litmus-test validation of the compound memory model (Table 5). The second reproduces the evaluation of lock-free data structures over BPF arenas (Table 6).

A Litmus-test Validation

A.1 Abstract

This artifact validates the compound C/LKMM model against the individual C and LKMM models via litmus testing (Table 5). Our executable model, `calculus.py`, evaluates each test's target execution as `Allowed` or `Forbidden`; agreement with `herd7` on the Linux kernel and Dartagnan RC11/C suites validates the preservation theorems, and generated cross-boundary variants validate the interchangeability theorem. The repository includes all source suites, their translations into our common calculus representation, and the translators. (It also includes Paul E. McKenney's RCU litmus tests as an optional suite exercising the engine's experimental RCU support; these are beyond the paper's claims.)

A.2 Artifact check-list (meta-information)

- **Algorithm:** litmus-test translation, execution-graph exploration, memory-model checking
- **Transformations:** source-to-source translation of Linux and RC11/C litmus tests into calculus litmus; implemented by included public Python translators
- **Data set:** Linux kernel litmus tests, Paul E. McKenney RCU litmus tests, RC11/C litmus tests
- **Run-time environment:** Linux/Unix-like command line, Python 3, local filesystem; no root access required
- **Output:** translated litmus file, final `Allowed`/`Forbidden` result
- **Experiments:** single-test evaluation, suite translation, suite validation
- **How much disk space required (approximately)?:** 100 MB
- **How much time is needed to prepare workflow (approximately)?:** 5–10 minutes
- **How much time is needed to complete experiments (approximately)?:** seconds for single tests; 5 minutes for baseline C and Kernel suite checks, 36+ hours for full validation of provided suites
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** GPL-2.0-only OR BSD-2-Clause
- **Archived?:** https://doi.org/10.5281/zenodo.23201428

A.3 Description

A.3.1 How to access. The artifact is accessible via the calculus subfolder of https://doi.org/10.5281/zenodo.23201428.

A.3.2 Data sets. Source suites are under `litmus/Kernel/` (Linux kernel), `litmus/paulmckrcu/` (Paul E. McKenney RCU), and `litmus/c/` (RC11/C). Translated tests are under the corresponding `converted/` directories.

A.4 Installation

All software dependencies are wrapped into a Dockerfile. Clone the repository and run `./scripts/run_docker.sh` while in the `calculus` subfolder.

A.5 Experiment workflow

All experiments are driven by a single script, `scripts/run_artifact_pipeline.py`. The simplest invocation runs the full single-language validation – the C suite and the Linux suite:

```
python3 scripts/run_artifact_pipeline.py --all
```

Each group can also be run separately via the `-suite [c/linux/RCU/interchange]` flag; the RCU suite is optional, exercising the engine's experimental RCU support beyond the paper's claims. E.g.,

```
python3 scripts/run_artifact_pipeline.py \
--suite interchange
```

Because it runs tens of hours, the interchange script can be run on a limited number of tests by using the `-stop-after [N]` argument and can be canceled early (Ctrl + C) to view a summary of results for the tests run so far.

Finally, a single litmus test can be run by providing a `.litmus` file, using the same formatting as `herd7`, as an argument. `-kind c/linux` specifies which memory model herd should use to compare against:

```
python3 scripts/run_artifact_pipeline.py \
path/to/test.litmus --kind [c/linux/converted]
```

A.6 Evaluation and expected results

The expected result is a `PASS` for each of the three groups: C validation, Linux validation, and interchangeability. The script prints a per-group summary; disagreements are reported as a list of failing tests, and none is expected.

A pass means the following. For the 3,272 C and Linux tests, the verdict produced by `calculus.py` matches that of `herd7` on every test. For the interchangeability tests, every variant of each test should have `calculus.py` return the same verdict as the baseline test from which it was generated (`herd7` has no compound C/LKMM model, so no external oracle exists for these). The optional RCU suite is likewise expected to match `herd7`; it exercises the engine's experimental RCU support and is not part of the paper's claims.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

B BPF Arena Data Structures

B.1 Abstract

To reproduce the data-structure benchmarks shown in Table 6. Each data structure is implemented as a two way relay, involving multiple concurrent threads for the userspace and kernel components. The benchmark harness records latency and throughput on both sides. This workflow reproduces (i) correctness: each data-structure runs correctly with no inconsistent data read by the consumer, (ii) performance: the SPSC structures perform in the same ballpark as one another, and faster than IO_URING.

B.2 Artifact check-list (meta-information)

- **Algorithm:** lock-free queues, SPSC rings, stack
- **Program:** self-contained microbenchmarks
- **Compilation:** Clang/LLVM 15+ with BPF target (`clang-20` recommended) and `gcc`; `libbpf` and `bpftool` are built from the included submodules
- **Binary:** built from source; x86-64 Linux
- **Run-time environment:** Linux 6.10+ configured with the included `kernel/configs/bpf-arena.config`; root access required to load and attach BPF programs
- **Hardware:** any x86-64 machine with 4 or more cores.
- **Run-time state:** sensitive — run as sole user
- **Execution:** root, sole user; roughly 10 seconds per data structure
- **Metrics:** per-op latency (ns), success rate, throughput
- **Output:** console tables, plus per-app logs under `build/`
- **Experiments:** `make`, then `scripts/usertests.py` and `scripts/runner.py`
- **How much disk space required (approximately)?:** 500 MB for the artifact; 20 GB more to build the kernel
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes to build the artifact; 1–2 hours to build / boot suitable kernel
- **How much time is needed to complete experiments (approximately)?:** 5 minutes
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** GPL-2.0-only OR BSD-2-Clause
- **Archived?:** https://doi.org/10.5281/zenodo.23201428

B.3 Description

B.3.1 How to access. The artifact is the `bpf-arena-data-structures` subfolder of https://doi.org/10.5281/zenodo.23201428. The data structures are in `include/`, the relay apps in `src/` as `skeleton_*.bpf.c` plus `skeleton_*.c` pairs, and the measurement infrastructure in `include/ds_metrics.h`.

B.3.2 Hardware dependencies. Our results are on Intel Core i5-7600 (4 core) with 16GB DDR4 memory. Any comparable multi-core machine will be sufficient, you can instantiate a d710 node on Cloudlab with Ubuntu 22.04 using the `profile.py` in the root of the `c-lkmm-cmcm-artifacts` repository.

B.3.3 Software dependencies. All software dependencies are wrapped into a Dockerfile and a Nix Flake. You may use either environment for this evaluation. We provide one-click installation scripts for both:

```
# Option 1: Nix
./scripts/install-nix.sh
nix develop

# Option 2: Docker
./scripts/install-docker.sh
docker build -t bpf-arena-ds .
```

Additionally, you will require Linux Kernel 6.18.44 with specific BPF configuration flags enabled.

```
./scripts/build-kernel.sh --base running -y
```

This will download kernel 6.18.44, apply the BPF config flags on top of the current config flags, build, and install the kernel.

Docker faces some restrictions here, refer to **Section 4** of the README.md.

You have to reboot after this step.

B.4 Run

Instantiate the same environment and build the data-structure suite.

```
nix develop
# OR
./scripts/init-docker.sh
```

The Docker script is required to configure BPF privileges in the container.

```
make
sudo ./scripts/run_all.py
```

This script verifies your kernel configuration, loads all required BPF related submodules, checks for suite binaries, tests the suite, and then runs the definitive benchmark on all data-structures.

B.5 Evaluation and expected results

The script above will run all the data structures to completion. The expected results will rank the performance of the data-structures: Folly SPSC, CK Ring SPSC, and CK FIFO SPSC will be in the same ballpark, performing faster than IO_URING.

```
========================================
PERFORMANCE RANKING (slowest first)
========================================
 #  Data structure      Ops/sec  E2E(ns)
```

Absolute values are not the reproducible claim. Producer latency may vary, and depends on core count, clock speed, and how fast the workload generates events.