
RCU Pseudo-Transactions

Bridging the gap between RCU and STM

Mathieu Desnoyers · EfficiOS

Linux Plumbers Conference 2026 · Prague · 6 October 2026

RCU's contract

No reader overhead. In exchange, a writer gets **exactly one** atomic operation: **a single pointer store**.



A pseudo-transaction does not add a fourth phase.
It **widens the second one**: N slots, still one store.

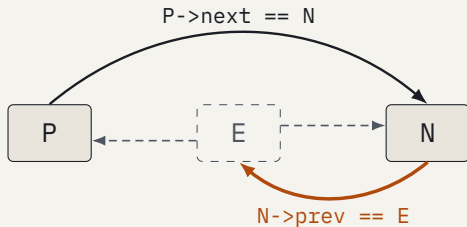
The kernel's list already needs two stores

```
static inline void __list_del(struct list_head *prev,
                             struct list_head *next)
{
    next->prev = prev;          /* N->prev: E -> P    plain store */
    WRITE_ONCE(prev->next, next); /* P->next: E -> N    marked store */
}

/* __list_add_rcu(): */
new->next = next;              /* private */
new->prev = prev;              /* private */
rcu_assign_pointer(list_next_rcu(prev), new); /* publishes new */
next->prev = new;              /* plain store, AFTER publish */
```

- Linux kernel and Userspace RCU rculist expose forward iterators only.
- Tradeoff: poison prev for use-after-removal debugging.

Publication atomicity, not ordering



No barrier fixes a window whose width is a grace period.

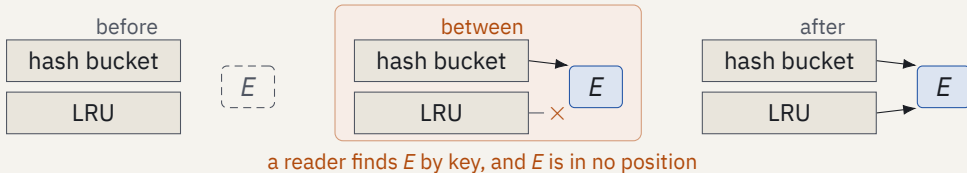
- Two edges are **two release stores**. Between them the forward chain has dropped E, the backward chain has not: **two different lists**, neither torn.
- E is a ghost until its grace period completes: the backward view trails by **up to a grace period**.

The kernel's narrow exception (Dec 2024): `list_bidir_del_rcu()` + `list_bidir_prev_rcu()` let a reader step back out of a ghost — not a coherent reverse walk.

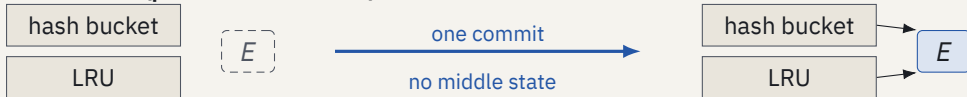
Composition: two structures, one object

A cache entry E must be **findable** (hash table) and **ordered** (LRU list).

Two publishes (plain RCU)



One commit (pseudo-transaction)



A writer-side lock does not help: **the observer is a reader, and takes no lock.**

Today's remedies each pay somewhere

Remedy	Who pays	How
Copy the region	the writer	allocation, footprint and reclaim on every update; a pinned node cannot be copied at all
Drain with a grace period	the writer	waits out every reader in the system
Steer or validate the reader	every reader	seqcount retries (dcache: d_seq, rename_lock), per-element locks: paid on every traversal
Decline the operation	the user	no RCU reverse iterator

What is missing is **publication atomicity** across slots, and RCU has no way to ask for it.

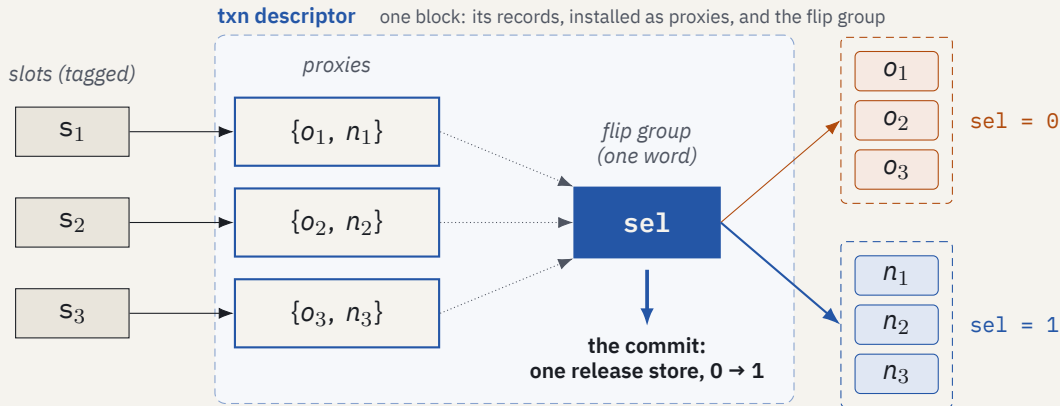
A pseudo-transaction

A **frozen** set of $\{slot, old \rightarrow new\}$ records. A sequence of **install** stores makes them visible, slot by slot, each still resolving *old*; **one atomic commit step** then publishes the transition to *new*, for all of them at once.

Journaling filesystem	Pseudo-transaction
1 journal transaction closed	record set frozen
2 journal blocks written	records installed: slots parked, still <i>old</i>
3 commit record written	one commit step: <i>old</i> $\rightarrow$ <i>new</i> , for all

Atomicity, not isolation: readers are plain RCU readers.

The flip-latch: one level of indirection



Every proxy reads one selector: **one release store flips the set.**

Lifecycle: only the commit is visible

Step	The slot holds	A reader resolves	Linearization point?
(before)	<i>o</i> , directly	<i>o</i>	—
Build	<i>o</i> , directly	<i>o</i>	no: new nodes unreachable
Install	tagged proxy, <i>sel</i> = 0	<i>o</i>	no: resolved value unchanged
Commit	tagged proxy, <i>sel</i> = 1	<i>n</i>	yes: one release store
Settle	<i>n</i> , directly	<i>n</i>	no: resolved value unchanged
(after)	<i>n</i> , directly	<i>n</i>	—

- Install and settle both rewrite the slot, and change nothing a reader resolves.
- Settle *is* the uninstall. Then `call_rcu()` frees the txn descriptor.

The read side is one branch

```
static inline void *urcu_txn_sw_resolve(void *v, uintptr_t tag)
{
    if (caa_likely(!urcu_txn_sw_is_proxy(v, tag))) /* (v & tag) != tag */
        return v;
    return urcu_txn_sw_proxy_get(urcu_txn_sw_untag(v, tag));
}

/* urcu_txn_sw_proxy_get(): */
proxy->ptr[atomic_load(&proxy->group->selector, CMM_ACQUIRE)]
```

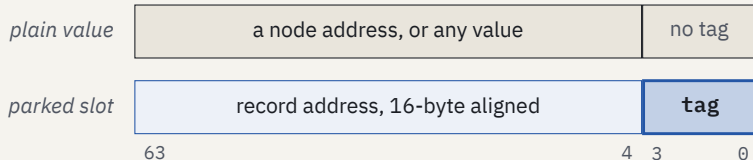
Not parked almost always

A test on a register already loaded, and a predicted branch. **No load.**

Parked install → settle, that slot only
 proxy → group → selector → ptr[sel]:
 indexed, a data dependency, not a branch.

The untag *subtracts* the tag, so the compiler folds it into the next load's displacement.

Tags: the marker rides the pointer



test: $(v \ \& \ \text{tag}) == \text{tag}$, on a value already loaded
record: $v - \text{tag}$, folded into the next load

- **Per-record tag:** one commit may span structures with different encodings.
- Records are 16-byte aligned: 4 low bits to choose from.
- **Contract:** no legitimate slot value may carry the tag pattern.

Exclusion is per slot

No internal locking. The embedder provides mutual exclusion **per slot**.

- **No install CAS, no conflict detection, no abort:** low writer overhead.
- Two writers on the same slot **silently corrupt** it. - `DURCU_TXN_SW_EXCL_VALIDATE` aborts the process on violation, with no overhead when disabled.
- With fine-grained locking (per bucket, per node), **writers on disjoint slots commit concurrently**.
- A commit can only fail on out-of-memory, before any slot is parked. Calling `urcu_txn_sw_reserve()` beforehand prevents it.

The API on one slide

```

struct urcu_txn_sw_txn txn;                                /* on-stack handle */
urcu_txn_sw_init(&txn);
urcu_txn_sw_reserve(&txn, 3);                               /* rules out OOM */
old = urcu_txn_sw_load(&txn, &a->next, TAG);                /* sees own writes */
urcu_txn_sw_record_chain(&txn, &a->next, old, x, TAG);
urcu_txn_sw_record(&txn, &b->prev, y, x, TAG); /* blind append */
if (urcu_txn_sw_commit(&txn) != URCU_TXN_STATUS_OK)
    /* URCU_TXN_STATUS_MEMORY_ERROR: nothing was published */;
/* reader, under rcu_read_lock(): */
v = urcu_txn_sw_resolve(rcu_dereference(*slot), TAG);

```

- `record()` appends blindly: slots pairwise distinct. `record_chain()` fuses onto this transaction's record for the slot: committed *old* kept, *new* advanced.
- One record: the commit is a plain release store.
- Two or more records: install, commit, settle, then `call_rcu()`.

What it provides, and what it does not

Provided

- **Write-set atomicity:** every record resolves *old* before the commit, *new* after.
- **No tearing:** each slot resolves to *old* or *new*, never an intermediate value.
- **The commit linearizes at one point** (transaction records only).
- **Cross-structure atomicity.** The unit is one pseudo-transaction: two commits are two linearization points.

Not provided

- **A reader is not a transaction of any kind:** a sequence of single-slot reads. A traversal may straddle a commit.
- **No opacity, no multi-read snapshot.**
- **No read validation:** mutual exclusion makes it unnecessary.

Never new-then-old

The selector is written once, $0 \rightarrow 1$, and never back. A reader that resolves the slots of one commit **along a dependency chain** sees

o o o n n n allowed n o never

The chain is an address dependency — every ordinary walk down a list or a tree — or an acquire.

New-then-old is what drives readers to seqcount retries. Here it cannot happen, by construction: no seqcount, no retry.

Between RCU and STM

	RCU	Pseudo-transaction	Opaque STM
Atomic to readers	one slot	N slots, one commit	read and write sets
The reader pays	nothing	one predicted branch no added load no per-element state	a barrier on each transactional read
The reader gets	one slot at a time	never new-then-old	isolation, opacity

Isolation is a read-side cost once updates mutate in place.
RCU pseudo-transactions decline to charge readers for it.

Honest asterisk: RCU gives readers a snapshot, existence a per-element membership test — strictly more, and charged to readers (slide 23).

The list: the reverse walk comes back

```
rcu_read_lock();  
urcu_txn_sw_list_for_each_entry_reverse_rcu(pos, &head, node)  
    ...;  
rcu_read_unlock();  
  
urcu_txn_sw_list_del_rcu(&e->node);    /* writer, under the embedder's locks */  
call_rcu(&e->rcu, free_e);
```

At every resolution, $a \rightarrow \text{next} == b \Rightarrow b \rightarrow \text{prev} == a$.

- A removed node keeps its own next and prev — a ghost — so a reader standing on it escapes either way.
- Coherence holds *at* each resolution. Two resolutions that must agree are asking for a snapshot, which is not provided.

What a list edit records: two edges, one flip

```
static inline int
urcu_txn_sw_list_add_after_prepare(struct urcu_txn_sw_txn *txn,
                                   struct urcu_txn_sw_list_node *newp,
                                   struct urcu_txn_sw_list_node *pos)
{
    struct urcu_txn_sw_list_node *next =
        urcu_txn_sw_list_pending(txn, &pos->next);
    newp->prev = pos;                /* build: unreachable */
    newp->next = next;
    urcu_txn_sw_record_chain(txn, (void **) &pos->next, next, newp,
                             URCU_TXN_SW_LIST_PROXY_TAG);
    urcu_txn_sw_record_chain(txn, (void **) &next->prev, pos, newp,
                             URCU_TXN_SW_LIST_PROXY_TAG);
    return 0;
}
```

The `_rcu` wrappers: `init` + `declare_disjoint` + `reserve(2)` + `_prepare` + `commit`.
Abridged from `rcu_txn_sw_list.h`: `(void)` casts and comments dropped, lines re-wrapped.

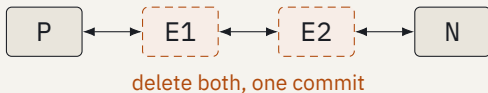
One commit, two structures

```
urcu_txn_sw_init(&txn);  
urcu_txn_sw_reserve(&txn, 3);           /* 1 hlist edge + 2 list edges */  
  
/* findable: */  
urcu_txn_sw_hlist_add_head_prepare(&txn, &e->hnode, bucket);  
/* ordered: */  
urcu_txn_sw_list_add_after_prepare(&txn, &e->lru, &lru.node);  
  
ret = urcu_txn_sw_commit(&txn);         /* one flip */
```

E becomes findable **and** ordered at the same instant. Hiding it has the same shape.

The abstract's composability claim: one object in several transaction-aware structures, made visible or hidden atomically. Slide 5's composition problem has no middle state left.

Read-your-own-writes



slot	old (committed)	new
P.next	E1	E2 → N fused
E2.prev	E1	P
N.prev	E2	P

The second delete must see the first one's *pending* P.next.

- `urcu_txn_sw_load()` returns this transaction's pending value.
- `record_chain()` fuses onto the record: keeps the committed old, advances the new.
- Find: a linear scan of a private array; a Bloom filter armed lazily for long write sets.

Without it, the second delete records E1.next on a node about to vanish.

Three obligations: search through the transaction's load, or before the first record; every anchor still live; `declare_disjoint()` is a promise (-DURCU_TXN_SW_DEBUG_DISJOINT traps a broken one). Read-your-own-writes itself is standard in deferred-update STM.

The hlist: one transacted edge

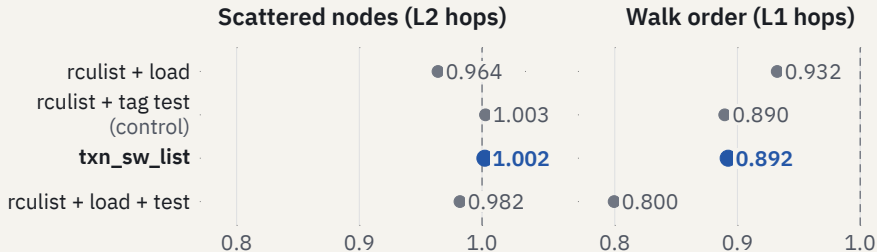
Readers walk an hlist forward only: **one** reader-visible edge per operation.

- `pprev` is only used by writers: plain stores.
- A single insert or delete costs about one `rcu_assign_pointer()`: no proxy, no `call_rcu()`.
- Composes with larger transactions through `_prepare` (slide 19).
- Single-pointer head: 8 bytes per bucket.

Reader: zero loads, one branch

Per node: `rculist` 8 instructions: 2 branches, 2 loads
`txn_sw_list` 10 instructions: 3 branches, 2 loads **+1 test, +1 branch, 0 loads**

Read throughput relative to `rculist` (dashed = 1.0), **higher is better**; 192 readers, no writer.



- **Scattered nodes**, as allocated over time: the branch is **free**, 1.002–1.003 of `rculist`.
- **Walk order, L1-resident**: it shows, 10.7% (the control: the same).

Setup: 2× AMD EPYC 9654 (192 cores), one thread per core; liburcu QSBR; 10 000 nodes; median of five 3 s runs.

Comparison with stronger reader guarantees

	Per-element state	Extra loads	Extra branches	Reader guarantee	Reader pays vs txn scattered / walk order
Existence	eh_egi field	1	1	per-element	2–3% / 15–17%
RLU	object header	1	1	snapshot	1–3% / 23–44%
MV-RLU	header + versions	≥ 2	≥ 3	snapshot	1.43× / 3.01×
This work	none	0	1	never new-then-old	—

They add reader overhead **and** provide additional guarantees: a tradeoff, not a defect.

Existence in its best layout (a 32-byte element); MV-RLU in its global-clock configuration, ratios at 192 readers. Write throughput is not ranked across this boundary.

Writer performance breakdown

1.3–1.8×

one writer, readers running

- Scattered nodes: 1.55–1.58× up to 4 readers, 1.6–1.8× beyond.
- Walk order: 1.34–1.39× up to 4 readers, 1.26–1.52× beyond.
- No reader: no ratio. Plain RCU outruns its own reclaim there.

Where the cycles go one writer, one reader, no mutex: the writer's thread, per update

rculist 129 cycles per update



txn_sw_list 243 cycles per update



write function
list op + record

call_rcu

nodes

commit

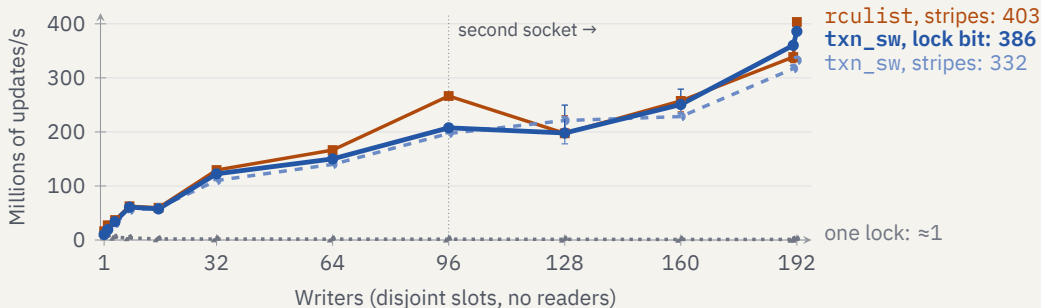
txn descriptor slab
allocate, hand back

Intrinsic: a fresh txn descriptor per update, kept for a grace period. The commit: 19 cycles of 243.

Not intrinsic: this slab. Under rseq: no lock, no atomics. A kernel port could use the allocator and `kfree_rcu()`.

Writers scale with fine-grained locks

Writers on disjoint slots: plan unlocked → lock in ascending address order → validate → commit.



Lock bit in each node: **10.3 → 386**, 38× from 1 to 192 writers, and 339× the list behind one lock.

rculist cannot use an in-node lock: its prev cannot be accessed coherently, so finding the predecessor takes a forward walk.

Dentry cache: bucket locks around SW commits

A **userspace model**, two engines.

The kernel's scheme

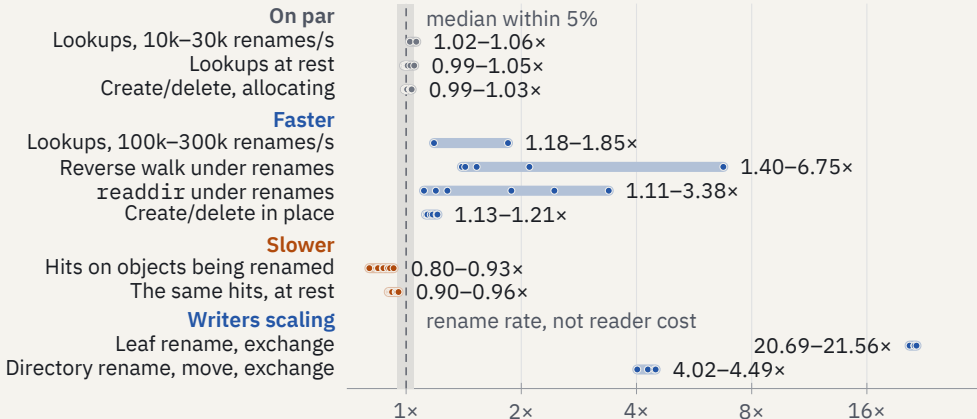
d_seq hand over hand,
rename_lock, i_rwsem.

Bucket lock + SW txn

Bit locks on the bucket and child-list heads. A rename is **one commit** across both indexes. No d_seq, no rename_lock, no lock in readdir.

A userspace model is not kernel evidence: no reference counts, and no i_rwsem, which filesystems rely on.

Dentry cache: results



Userspace model. Reader rows: both engines at the same paced rename load.

What a kernel port needs

Needs

- A spare low bit in the slot: list pointers are aligned.
- A commit fails only on OOM, before anything is parked. Reserve before the first irreversible step and it cannot fail — under a spinlock, say.
- One `kfree_rcu()` per multi-edge commit: it batches already.
- The exclusion validator as a debug option.

Where it would land

- A coherent reverse walk for rculist: the iterator it declines today.
- Cross-structure publish: hash + LRU.
- Pinned objects whose back edges must move in place.

For the dentry cache, also

- Reference counts, or hazard pointers?
- `i_rwsem`: filesystems rely on it.

What it is not

- **Not STM.** No snapshot, no opacity: a reader is not a transaction of any kind.
- **The embedder provides mutual exclusion**, per slot. Two writers on one slot corrupt it.
- **The tag contract:** no legitimate slot value may carry the tag pattern. The contracts underwrite memory safety, not just atomicity.
- **Commit width grows with edges:** a tall tower commits many records where existence flips one group.
- **The writer pays 1.3–1.8×**.

Summary

- RCU gives a writer one atomic store. A pseudo-transaction widens it to **N slots, still one store**: the flip-latch.
- Readers: **no added load, no per-element state**; one predicted branch, free on scattered nodes.
- Writers: 1.3–1.8×, mostly the txn descriptor's slab, not the flip. They scale with the embedder's fine-grained locks.
- Never new-then-old, and no more: **not STM**.

Paper efficios.com/pub/rcu-txn/mathieu-desnoyers-rcu-pseudo-transactions.pdf

Code github.com/compudj/userspace-rcu-dev @ 2793224e, in include/urcu/:
rcu-txn-sw.h, rcu-txn-sw-list.h, rcu-txn-sw-hlist.h, rcu-txn-status.h

Questions?

Anticipated questions

Why not a seqcount?

A read-side retry on every traversal, detecting after the fact. Here a reader never retries — and gets no snapshot either.

Why not copy?

The cost scales with the region, and a pinned node cannot be copied.

Isn't this existence?

Existence adds a per-element field (+1 load, +1 branch) and gives a stronger per-element guarantee. Pseudo-transactions add no per-element state; existence's readers pay 3% of read throughput on scattered nodes, 15% in walk order.

ABA?

Resolution goes through the descriptor, not the slot value: slot-value A-B-A is inert.

HTM?

Commodity HTM may abort, so it needs a fallback that meets the same problem. s390 constrained transactions: at most 32 instructions in 256 bytes.

Does the tag test need a barrier?

No. The proxy's immutable fields are reached through the dependency; only the selector is loaded with acquire, and only when parked.

What if two writers race?

Silent corruption, by contract. -DURCU_TXN_SW_EXCL_VALIDATE catches it.

Isn't this MCAS?

The descriptor-and-status shape is: Harris et al. (2002). Nearest, Guerraoui et al. (2020): $k+1$ CAS, unlock deferred to reclamation. Here every step is a plain store, and settle is eager.

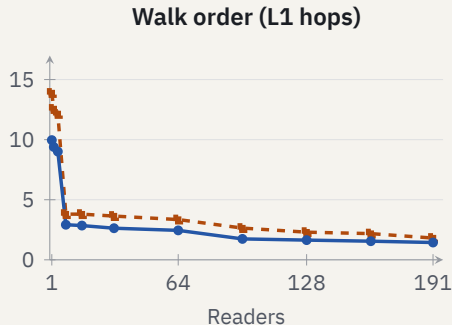
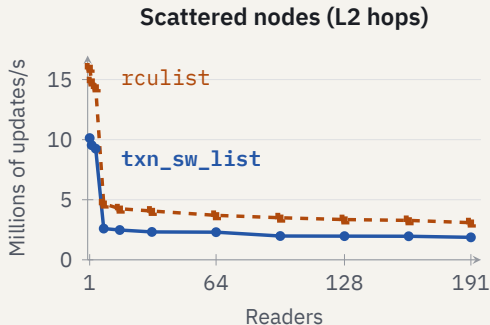
Split edits: publish in reverse read order

When one edit must span several commits, commit in the **reverse** of the reader's read order.

- Reader resolves $s_1, \dots, s_k$ in that order; the writer publishes s_k first, s_1 last. Once a reader sees one slot new, every later slot is new too: a prefix of old, a suffix of new.
- Example: re-parent E from P to Q . Commit $E \rightarrow \text{parent}$ first, then the child links. A reader that descends into E under Q then finds $E \rightarrow \text{parent} == Q$.
- Needs one fixed read order and distinct slots. A list read in both directions has none to reverse: it needs the atomic commit.

First stated, to our knowledge, by Triplett. An embedder's discipline: the facility cannot know a structure's traversal order.

One writer, with readers



Both writers collapse between four readers and eight. Scattered: the ratio is 1.55–1.58 up to four readers and 1.61–1.80 beyond. Walk order: 1.34–1.39 up to four readers, 1.30 at eight, 1.26–1.52 up to 191.

Dentry cache: the model, and where it loses

The baseline is the kernel's

- Lookup: the RCU walk. `d_seq` hand over hand; `rename_lock` sampled at `path_init()`, consulted on a miss.
- Reverse walk: a port of `dentry_path_raw()`.
- Locks: the kernel's `rw_semaphore` as `i_rwsem`, bit locks that spin as `bit_spin_lock()` does, a queued spinlock on the LRU shards.

The comparison

- Readers are compared under paced writers, at the points both engines' writers sustain.

Not modelled

- Reference counts (`lockref`): walks stay in RCU. Mounts, hard links, security hooks.

How the model commits

- `rcu-txn.h`'s single-writer commit: park, one flip, settle. No CAS, no contention abort.
- Not `rcu-txn-sw.h`: readers resolve through the commit's descriptor, not a proxy's group.

Where it loses

- Hits on objects being renamed: 0.80–0.93×. A rename publishes a shell; until a `call_rcu` callback folds it back, about 5 ms, a hit reads three cache lines instead of one.
- At rest: 0.90–0.96×, per-hop pointer decoding.

What the writers' lead includes

- The baseline takes `i_rwsem`, which filesystems rely on, and `s_vfs_rename_mutex`; the model neither. A port gains only what the VFS stops taking.