
Kernel Lock Contention Hotspots Causing Frame Drops on Android Mainline

From frame-drop attribution to optimizing three contended kernel locks

Presented by Bo Zhang

Co-authors: Barry Song, Hongru Zhang, Bo Zhang · Xiaomi Corporation

Linux Plumbers Conference 2026 · Android Microconference

Why Kernel Lock Contention Matters for UI

- At 90 Hz, each frame has only a ~11 ms budget.
- When a UI-critical thread (e.g. RenderThread) blocks inside the kernel, the frame is likely dropped.

How we measured

- Pixel 6, Android 17, Linux 7.1.0-mainline, 90 Hz refresh.
- 17 popular Android apps profiled.
- ftrace contention_begin/contention_end events correlated with Perfetto FrameTimeline to establish causal links.

~4.7%

of all dropped frames are causally attributable
to kernel lock contention

Test model: Repeated UI Scrolling under background residency pressure

The model combines repeatable user-visible interactions with realistic multi-app system pressure, while capturing frame behavior and kernel lock contention in the same test window

EXECUTION MODEL

```
for round = 1..10
  for app in 17 foreground apps
    start with target and background apps closed
    launch the fixed 20-app background set
    launch target; open its predefined page
    repeat 5 times
      scripted scrolling
      collect Perfetto + eBPF lock traces concurrently
    close target and all 20 background apps
```

TEST MATRIX

Foreground apps	17
Background set	20 apps
Rounds	10
Repetitions / app / round	5
Total test repetitions	1,000
Lock-wait threshold	≥ 3 ms

BACKGROUND STATE

Popular applications and system applications are pre-launched and retained in the background to reproduce realistic memory, scheduling, and asynchronous activity pressure.

FOREGROUND INTERACTION

17 popular applications covering social feeds, short video, e-commerce, local services, media, live streaming, and navigation. Each app is exercised on one predefined page.

OBSERVATION

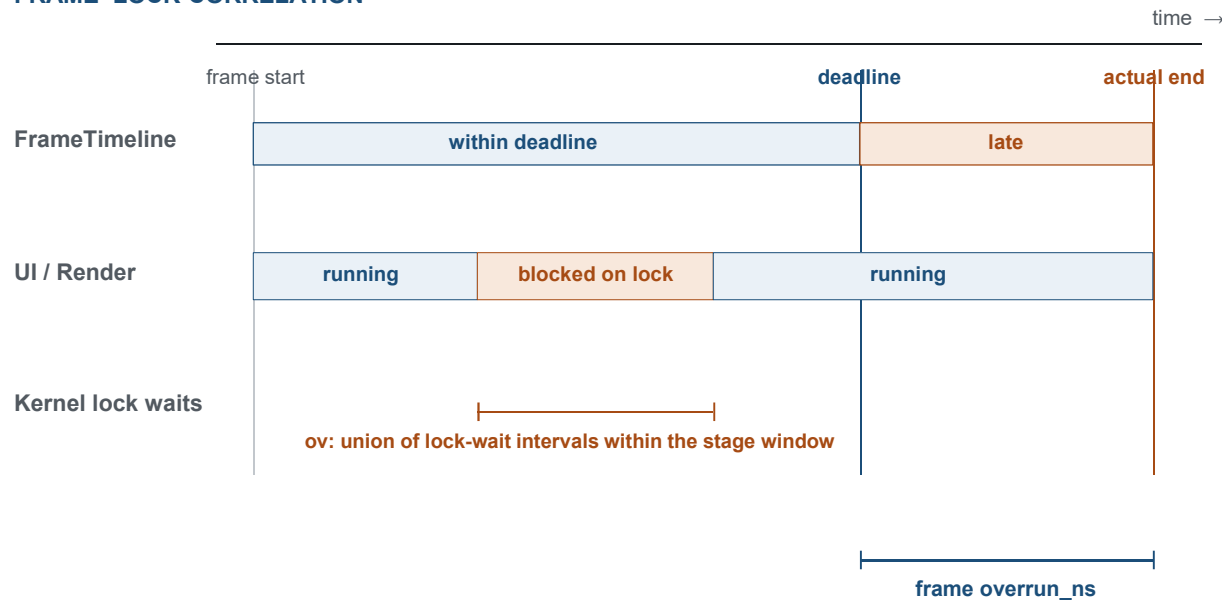
Perfetto — frame timeline, scheduling and system activity
eBPF — completed lock waits ≥ 3 ms
events: lock:contention_begin/end

TEST DEVICE **Google Pixel 6** 8 GiB · CPUs 0–7 · Android 17 · Linux 7.1.0-mainline

Attributing janky frames to kernel lock contention

Each janky frame is scored based on where lock contention is observed and how much lock-wait time contributes to the frame's deadline overrun

FRAME-LOCK CORRELATION



```
overrun share = min(1, ov / overrun ns)
```

Overlapping lock-wait intervals are merged and counted once.

Direct evidence uses the UI/Render window; indirect evidence follows scheduler wakeups (≤ 3 hops).

LOCK EVIDENCE PATH

Lock evidence path	ceiling
UI/main thread blocked during doFrame	10
RenderThread blocked during DrawFrames	9
scheduler wakeup chain (≤ 3 hops) $\rightarrow$ app thread	6
scheduler wakeup chain $\rightarrow$ SurfaceFlinger	5
scheduler wakeup chain $\rightarrow$ HWC/composer	4
scheduler wakeup chain $\rightarrow$ kernel thread	3

SCORE

```
score = round(ceiling × [0.3 + 0.7 ×
    overrun share])
```

The evidence path selects the ceiling; ov / overrun_ns scales it from 30% to 100%.

No lock path found → score = 0

FINAL DECISION	score ≥ 6	counted as lock-caused jank	score < 6	not counted as lock-caused jank
lock-caused jank	yes	yes	no	no
not lock-caused jank	no	no	yes	yes

With ceilings of 5, 4, and 3, SurfaceFlinger-, HWC-, and kernel-thread-only paths cannot reach the ≥ 6 threshold.

Top 5 Contended Subsystems (> 90% of lock-caused drops)

1	Mali GPU Driver	Job submission / scheduling locks block RenderThread	Open for discussion
2	mmap_lock	Read/write contention: page faults vs mmap/mprotect	Patch — covered today
3	ZRAM Compression	Priority inversion in swap-in path, > 20 ms	Patch — covered today
4	Binder IPC Allocator	Buffer allocation lock contention, > 8 ms	Patch — covered today
5	cgroup threadgroup rwsem	fork / exit path contention, > 11 ms	Open for discussion

Today we focus on the three mm / IPC locks that already have RFC patches.

mmap_lock: read/write contention on page faults

Key idea: Keep more faults on the lightweight per-VMA lock, avoid waiting on a write-contended mmap_lock.

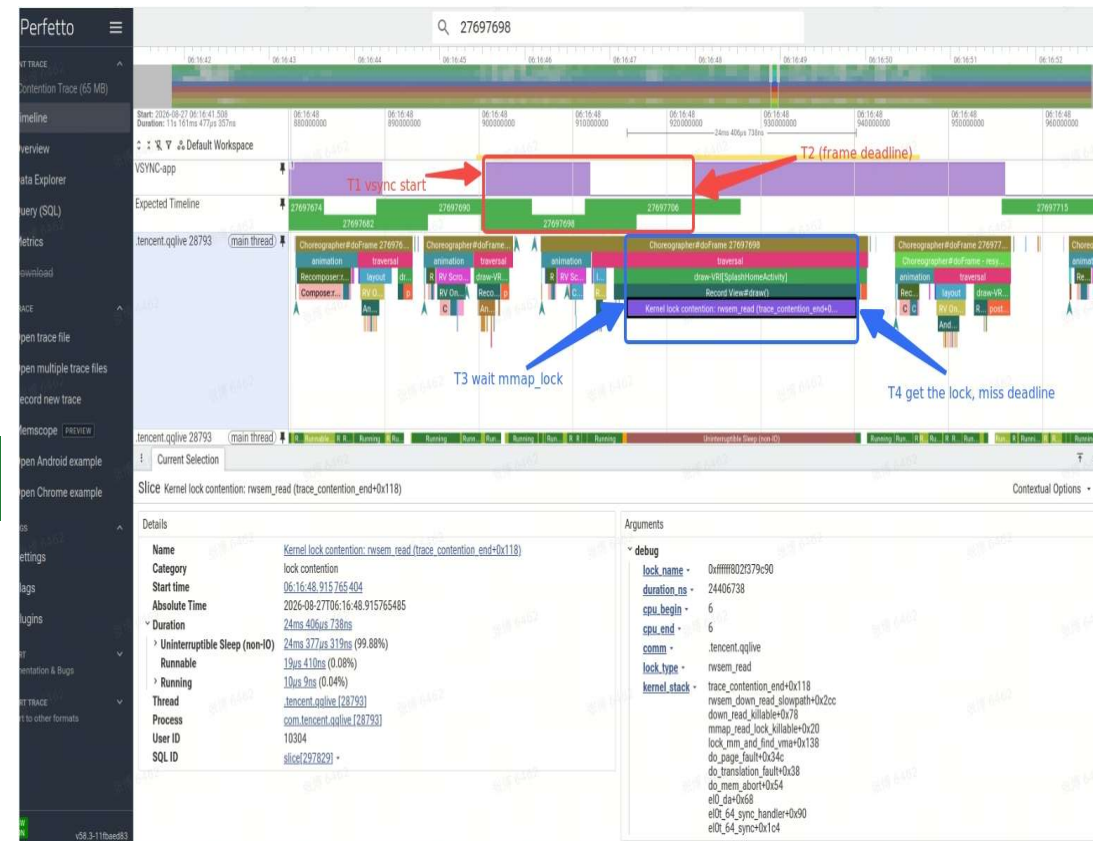
PROBLEM

Priority inversion on mmap_lock:

1. A reader holds the read lock, blocked on I/O.
2. A writer (mmap / mprotect) waits for the write lock.
3. Later readers — incl. the UI main thread — queue behind it and stall.

SOLUTION

- Retry once under the per-VMA lock in the arch fault handler before falling back to mmap_lock.
- The first fault already waited for the folio, so the retry usually succeeds.
- ~8 lines per arch, no core mm changes.



mmap_lock: microbenchmark results

mmap_lock contention count (lower is better) — Intel i7-12700, filemap benchmark

Threads	Vanilla	Patched	Reduction
40	3,187,336	52,086	-98.4%
60	4,385,154	65,079	-98.5%
80	5,337,890	69,708	-98.7%

- Swap throughput under 8 mmap writers: 0.58 /s → 15,420 /s (vanilla nearly stalls; patched holds up).

mmap_lock: real-world Android results

Two user-visible consequences — impact varies a lot across apps:

1 · Cold app startup — slower launch

Vivo, Douyin (TikTok China) cold launch: **-15.5%** (666 → 563 ms, SM8975, avg of 100 runs).

2 · Runtime — dropped frames

Main thread stalls on mmap_lock past the vsync deadline (see trace on previous slide).

Main-thread mmap_lock contention — 17 popular apps, scroll/swipe, 10 runs each

App	Contention events (Vanilla → Patched)	Total contention time (Vanilla → Patched)	Time reduction
Overall (17 apps)	722 → 235	12,315 ms → 4,284 ms	-65% time
Tencent Video	158 → 17	2,452 ms → 177 ms	-92.8%
JD.com	80 → 32	3,145 ms → 1,172 ms	-62.7%
Douyin	41 → 6	572 ms → 95 ms	-83.4%
NetEase Music	50 → 0	521 ms → 0 ms	-100%

- All 17 apps improved; 12 of them by > 60% in contention time. Real apps on real devices — not synthetic benchmarks.

ZRAM compression: priority inversion on swap-in

Key idea: Never hold the stream mutex across a sleepable / direct-reclaim allocation — the bounce buffer lets us drop it safely.

PROBLEM

Since compression became sleepable & preemptible (commit 2efa9e9eb4db):

The stream mutex can be held while its holder sleeps:

- (a) preempted / migrated to a little core, or
- (b) direct-reclaim `zs_malloc()` sleeps under the mutex.

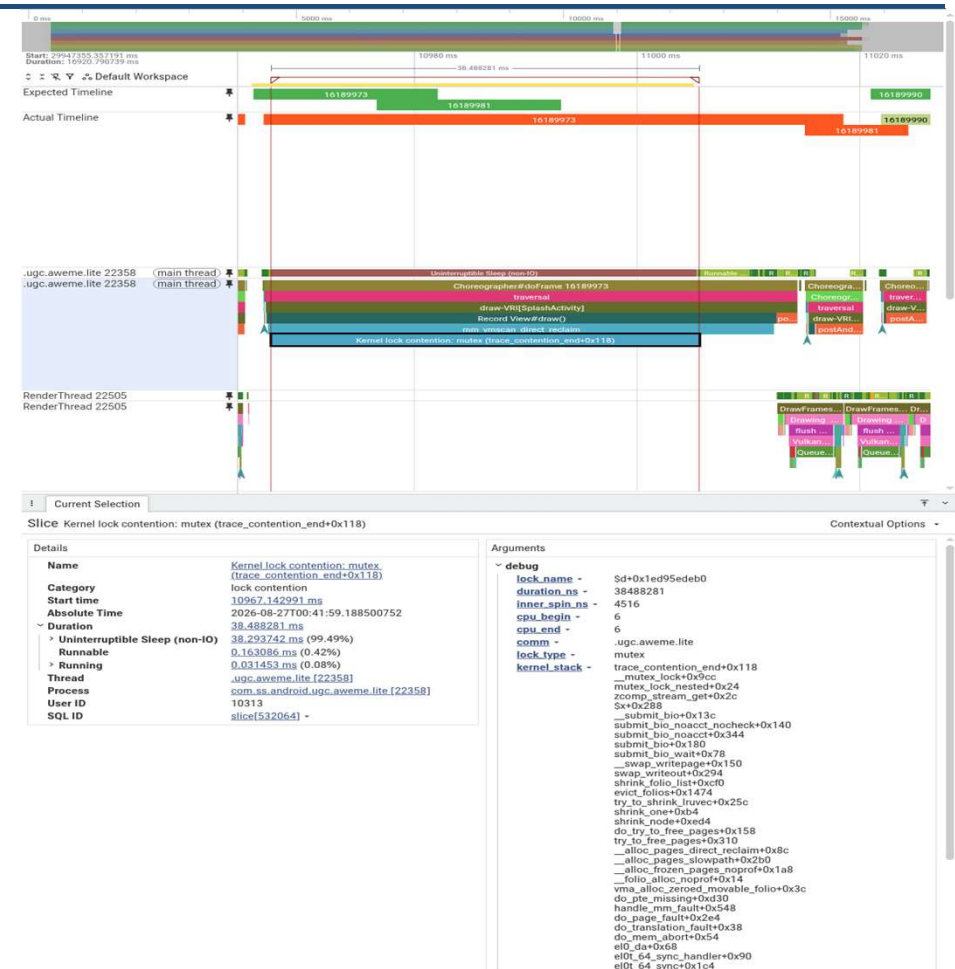
→ a high-priority thread (RenderThread) stalls > 20 ms on swap-in.

SOLUTION

Part 1 · sync backends: `preempt_disable()` around the stream (covers a).

Part 2 · don't sleep under the mutex (covers b):

try a non-sleepable `zs_malloc()` first;
on failure, copy to a bounce buffer, drop the mutex, then do the sleepable alloc — no re-compression.



ZRAM compression: the failure mode

Priority inversion: the stream mutex is held while sleeping

BIG core
high-priority

RenderThread needs swap-in → BLOCKED, waiting for stream mutex (> 20 ms)

▼ *waits on the same mutex held by a task that sleeps*

mutex holder
sleeps

(a) preempted / migrated to a little core, or
(b) sleeps inside a direct-reclaim `zs_malloc()` — still holding the mutex

Fix (two parts):

- `preempt_disable()` for sync backends → covers cause (a).
- Bounce buffer + drop the mutex before the sleepable `zs_malloc()` → covers cause (b), with no re-compression.

Binder allocator: contended alloc->mutex

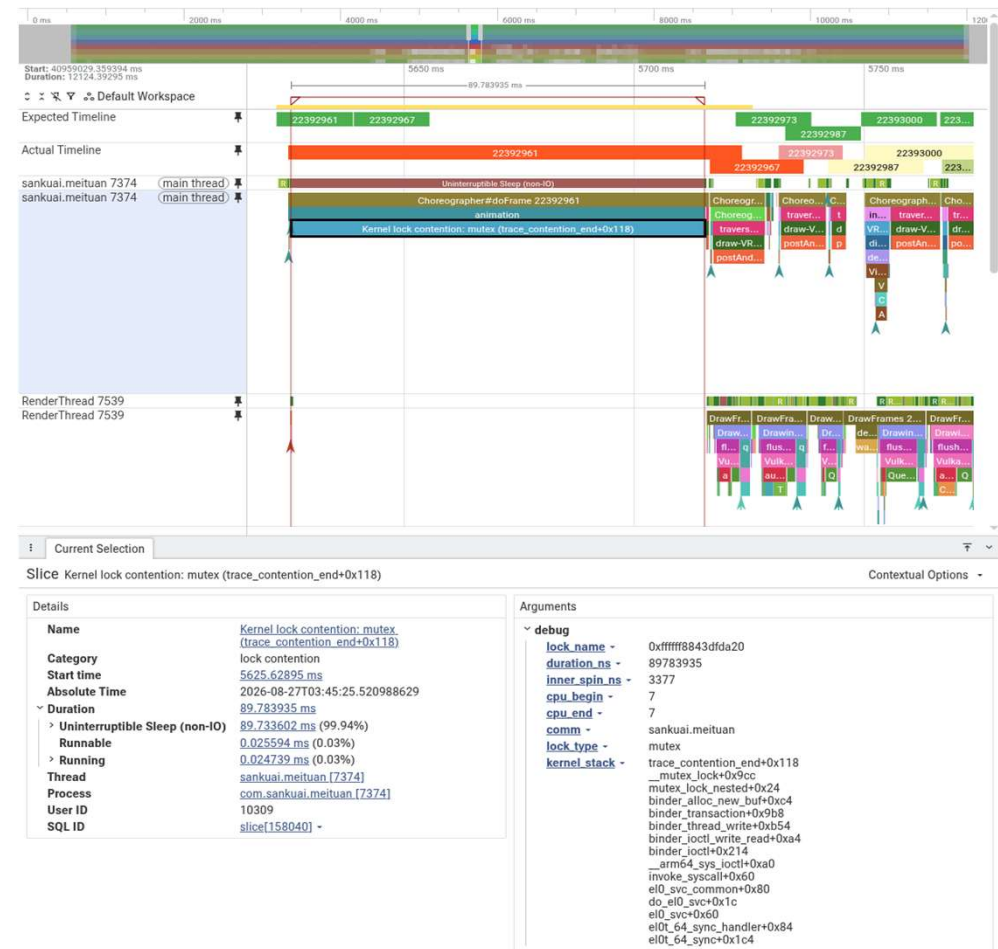
Key idea: Removing the sleeping lock from the hot path eliminates the priority inversion; shrinker uses trylock to avoid self-deadlock / ABBA.

PROBLEM

- alloc->mutex is highly contended on Android devices.
- When a low-priority task holds this mutex and sleeps, high-priority binder transactions are blocked.
- Result: priority inversion and latency spikes — buffer allocation contention observed > 8 ms.

SOLUTION

- Split the mutex into two locks with distinct roles.
- alloc->lock (spinlock): the hot, non-sleeping metadata — pages[], LRU list, rb-trees, free_async_space.
- install_mutex: only serializes the sleeping PTE operations (page install vs shrinker zap).



Binder allocator: results

binderThroughputTest — Qualcomm SM8850, 2 workers, 10 runs, under concurrent drop_caches

Metric	mutex (baseline)	spinlock + install_mutex
Throughput	27k–59k iter/s	84k–89k iter/s
Average latency	0.031–0.068 ms	0.021–0.022 ms
P99 latency	0.088–0.148 ms	0.046–0.058 ms

- Tested with KASAN (HW_TAGS), PROVE_LOCKING and DEBUG_ATOMIC_SLEEP enabled — no warnings.
- P99 latency roughly halved; throughput up ~1.5–3×.

Summary & Discussion

Common theme: lock contention → priority inversion / read-write exclusion → tail latency → dropped frames.

mmap_lock

Shorten the fallback path — retry once on the lightweight per-VMA lock before mmap_lock.

ZRAM

Remove priority inversion — disable preemption for synchronous compression backends.

Binder

Split the lock — move the hot path to a spinlock, isolate sleeping work under a separate mutex.

To discuss with the community

- How effective are the current mitigations in practice? What remains to be addressed?
- Mali GPU driver and cgroup_threadgroup_rwsem — known issues? Directions to reduce contention?

Thank you — Questions?

RFC patches

mmap_lock: lore.kernel.org/lkml/20260911025613.1220845-1-zhanghongru@xiaomi.com

ZRAM: lore.kernel.org/linux-mm/20260805005545.66112-1-baohua@kernel.org

Binder: lore.kernel.org/lkml/20260908145854.1130776-1-zhangbo0325@gmail.com

Backup

Code details & methodology — shown on demand during Q&A

mmap_lock — the retry (arch fault handler)

```
lock_vma:
    vma = lock_vma_under_rcu(mm, addr);
    if (!vma)
        goto lock_mmap;
    ...
    /* fault handled under per-VMA lock */

+   if (!vma_lock_retried) {
+       vma_lock_retried = true;
+       goto lock_vma;      /* retry once, still on per-VMA lock */
+   }

lock_mmap:          /* only now fall back to mmap_lock */
retry:
    ...
```

- Same ~8-line change replicated across arm, arm64, loongarch, powerpc, riscv, s390, x86.
- No changes to core mm code.

ZRAM — the two-part fix

```
/* Part 1: sync backend can't be preempted while holding stream */
struct zcomp_strm *zcomp_stream_get(struct zcomp *comp) {
    mutex_lock(&zstrm->lock);
+   if (!comp->ops->async && !IS_ENABLED(CONFIG_PREEMPT_RT))
+       preempt_disable();
}

/* Part 2: never sleep under the mutex during allocation */
+static unsigned long zram_zs_malloc(..., struct zcomp_strm *zstrm,
+                                     size_t comp_len, void **bounce) {
+   handle = zs_malloc(..., __GFP_KSWAPD_RECLAIM ...); /* no direct reclaim */
+   if (!IS_ERR_VALUE(handle)) return handle;
+
+   *bounce = kmalloc(comp_len, GFP_ATOMIC); /* preserve compressed data */
+   memcpy(*bounce, zstrm->buffer, comp_len);
+   zcomp_stream_put(zstrm); /* DROP the mutex */
+   handle = zs_malloc(..., GFP_NOIO ...); /* now safe to sleep */
+   return handle;
+}
```

- Bounce buffer avoids re-compression — the compressed bytes are copied out first.
- async flag added to struct zcomp_ops; false for almost all CPU backends.

Binder — split lock + shrinker trylock

```
struct binder_alloc {  
-     struct mutex mutex;  
+     spinlock_t    lock;          /* hot metadata */  
+     struct mutex   install_mutex; /* sleeping PTE ops */  
     ...  
};  
  
/* hot path: no more sleeping lock */  
-     mutex_lock(&alloc->mutex);  
+     spin_lock(&alloc->lock);  
    buffer = binder_alloc_new_buf_locked(...);  
  
/* shrinker must not block -> avoid self-deadlock & ABBA */  
+     if (!mutex_trylock(&alloc->install_mutex))  
+         goto err_get_install_mutex_failed;
```

- pages[] and LRU always updated together under alloc->lock → consistent state.
- deferred_release drops alloc->lock around the sleeping memset / page free.

Methodology detail

- Device: Pixel 6, kernel 6.18.0-mainline, 90 Hz.
- Workload: 17 popular Android apps.
- Signals: ftrace lock_contention events + Perfetto FrameTimeline.
- Attribution: correlate a contended lock event on a UI-critical thread with a frame that missed its deadline.
- Result: ~4.7% of dropped frames attributable to lock contention; 5 subsystems account for > 90% of those.