

Microseconds Matter

Benchmarking Thread Synchronization on Linux

Gautham Ponnu

What You'll Walk Away With

A latency floor

How fast can you go with zero kernel involvement — and what entering the kernel actually costs.

One result that inverts

The primitive that's fastest without RT becomes the worst under RT. You'll see why.

Tail latency, not just median

p99, p99.9, and max under idle, same-CPU load, and cross-CPU load. Where the surprises are.

A rubric

Concrete guidance for picking the right primitive — and when the primitive isn't the problem.

What We Measure

Defining Signal-to-wakeup latency

The time from the moment a signaler records its timestamp (just before notifying) to the moment the waiter records its own timestamps (just after returning from the blocking call).

```
latency_ns = t_wake - signal_time
```

Timestamps from

```
clock_gettime(CLOCK_MONOTONIC, ...) .
```

`nsec` resolution on both platforms.

Statistics collected :

Metric	What it tells you
p50	Typical case
p95	Common-ish outlier
p99	Rare outlier
p99.9	Worst of 1000
max	Absolute worst
stddev	Jitter / spread

Why Does It Matter?

When building real-world systems which are concurrent, thread wakeup is inevitable. Multiple primitives exist which all effectively do the same thing.

The scheduler will wake your thread.

Let's try to answer the question of **when, exactly?**

P.S.: Core Isolation is intentionally excluded.



$$\text{latency_ns} = \text{t_wake} - \text{signal_time}$$

Measurement: `CLOCK_MONOTONIC`, nanosecond resolution

What Affects Latency

Every measured number reflects hardware, kernel, and scheduling factors. Some we controlled tightly — others we intentionally let vary to observe real-world effects. Here is what fell into each bucket.

✓ CONTROLLED

Factor	How
CPU frequency	<code>performance</code> governor · fixed MHz
Warmup	100 iterations discarded
Load conditions	idle · same-cpu · diff-cpu
Scheduling policy	SCHED_OTHER vs SCHED_FIFO:80
Inter-trial cache state	base · clock-sleep · memset flush
Sample depth	1 M iterations per run

⚠ NOT CONTROLLED

CPU pinning — threads float across CCDs. Cross-CCD wakeups traverse Infinity Fabric (~2× slower). OS migration mid-run inflates tail and max.

Cache coherency — MESI latency depends on thread placement. Spinning primitives measure this directly; sleeping primitives embed it in the kernel path.

C-states — not disabled. Frequency pinning prevents deep sleep; shallow C-states between iterations remain possible.

IRQ affinity — system interrupts can land on benchmark CPUs and cause preemption.

Kernel version (RT vs standard) — different source trees (6.6.135-lts66 vs 6.6.87-rt54). Scheduling policy differences dominate.

Test Systems

PRIMARY — X64

CPU	Ryzen 7 3700X · 16 threads
Freq	3600 MHz
standard	Linux 6.6.135-lts66 · SCHED_OTHER
RT	Linux 6.6.87-rt54 · SCHED_FIFO:80
Runs	6 conditions · 1 M samples each

SECONDARY — AARCH64

CPU	Cortex-A72 · 4 cores
Freq	1500 MHz
standard	Linux 7.0.3-aarch64 · SCHED_OTHER
RT	Linux 6.18.29-v8-rt+ · SCHED_FIFO:80
Runs	6 conditions · 1 M samples each

The RT and standard x64 kernels differ slightly. Scheduling policy differences dominate over kernel version.
aarch64 diff-cpu results are confounded: with only 4 cores, benchmark (CPUs 0–1) + load (CPUs 2–3) leaves no idle capacity — not a fair isolation test.

Primitives on the Menu

Spinning — the floor

Primitive

Thread while waiting

`busy_wait_atomic`

per-waiter cache-line spin +
`PAUSE`

`userspace_rcu`

spin under
`rcu_read_lock/unlock`

Latency = cache-coherency propagation only. No kernel involved.

Sleeping — built on top of the kernel

Primitive

Thread while waiting

`condvar`

blocked in kernel

`pthread_condvar`

blocked in kernel

`eventfd`

blocked in kernel (fd)

`posix_sem`

blocked in kernel

`futex`

blocked in kernel (or races to
skip it)

1 M iterations · 2–16 threads · 6 load conditions each

busy_wait_atomic

The latency floor — a pure spin with no kernel involvement whatsoever

How `busy_wait_atomic` Works

Each waiter spins on its own **cache-line-isolated**

```
std::atomic<uint32_t> . cpu_relax()
```

(`PAUSE` / `YIELD`) prevents pipeline stalls on the spin.

What we measure: pure cache-coherency propagation — time from the signaler's release-store to the waiter's acquire-load. No kernel involved.

This is pretty much the **floor** every other primitive is measured against.

SIGNALER

```
while (ack_count.load(acquire) != 0)
    cpu_relax();

ack_count.store(n_waiters, relaxed);
clock_gettime(CLOCK_MONOTONIC, &signal_time);
for (w = 0; w < n_waiters; ++w)
    sig[w].gen.store(gen, release);
```

WAITER (PER THREAD W)

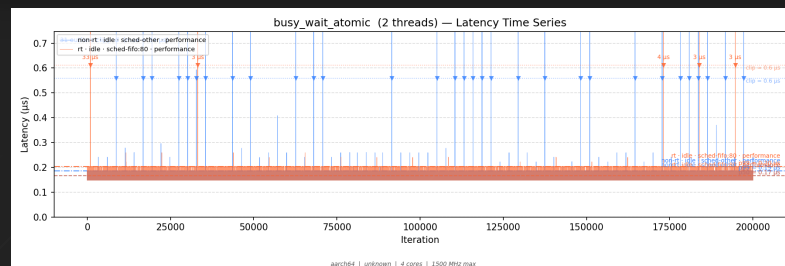
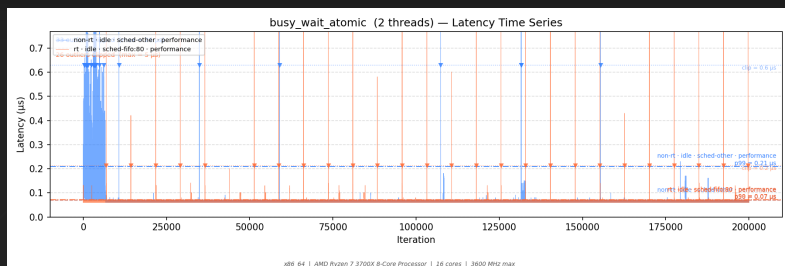
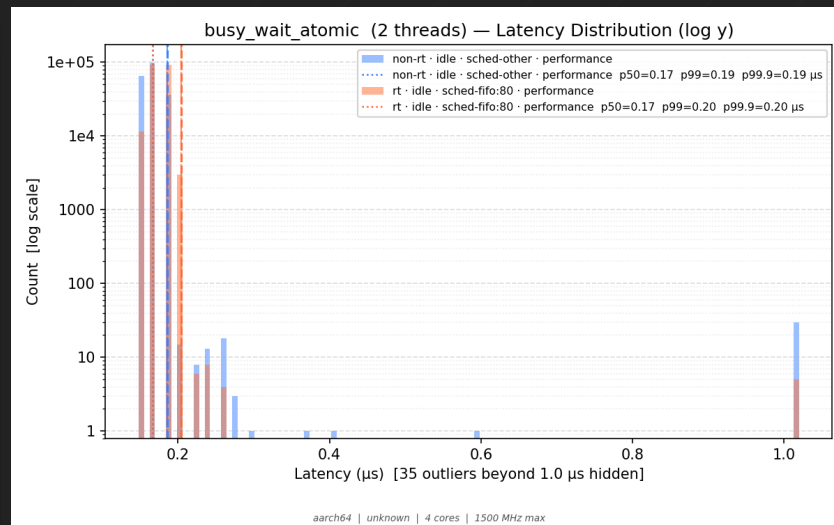
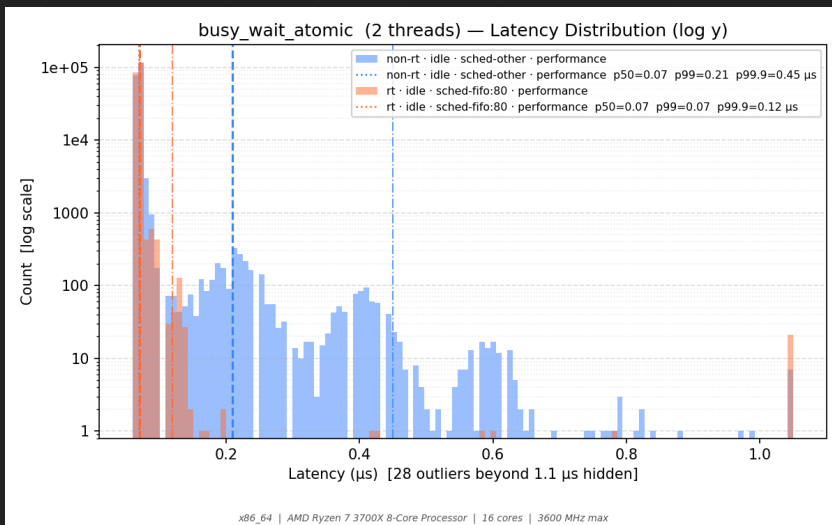
```
while (sig[w].gen.load(acquire) != expected)
    cpu_relax(); // PAUSE / YIELD

clock_gettime(CLOCK_MONOTONIC, &t_wake);
lat = diff_ns(signal_time, t_wake);
ack_count.fetch_sub(1, release);
```

busy_wait_atomic — x64 vs aarch64

x64

aarch64



2t · idle · 1 M samples · histogram (main) · timeseries (spike pattern)

userspace_rcu

Spin under `rcu_read_lock/unlock` — measures cache-coherency latency, not wakeup latency

How `userspace_rcu` Works

urcu-memb flavor: the waiter busy-polls a generation counter inside `rcu_read_lock/unlock`, which emit memory barriers. No kernel involvement on the signal path.

What we measure: cache-coherency propagation plus `rcu_read_lock/unlock` barrier overhead — similar to `busy_wait_atomic` but with the extra barrier cost.

The cost: 100% CPU while waiting. Under SCHED_FIFO:80 a spinning thread monopolizes its CPU, causing rare but extreme outliers.

SIGNALER

```
for (w = 0; w < n_waiters; ++w)
    sem_wait(&state.ack);

clock_gettime(CLOCK_MONOTONIC,
              &state.signal_time);
state.gen.store(i + 1,
               memory_order_release);
```

WAITER (PER THREAD W)

```
rcu_read_lock(); // reader: compiler barrier + epoch store, no syscall

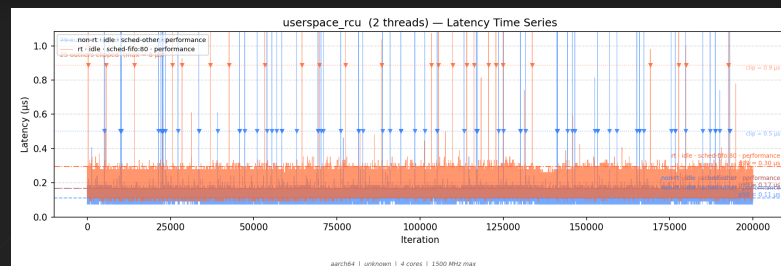
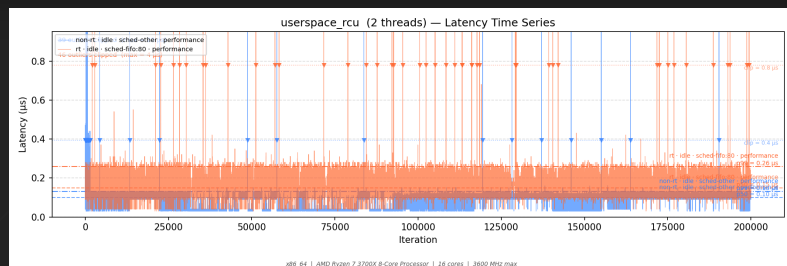
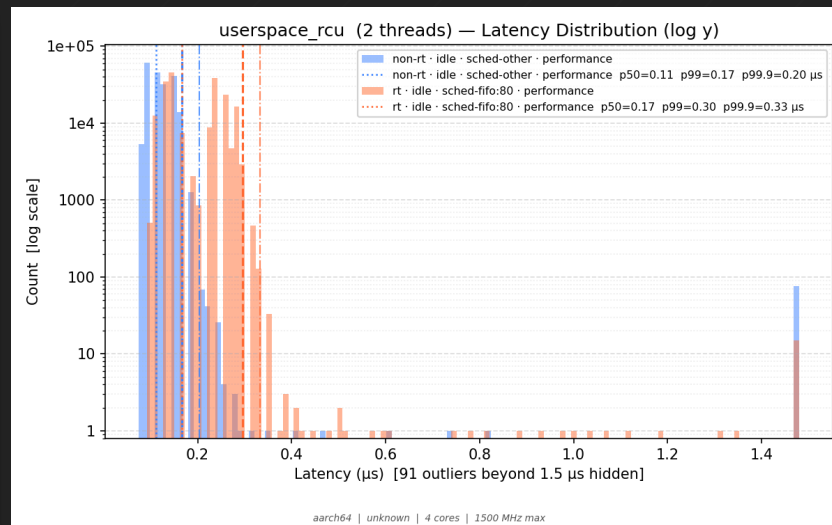
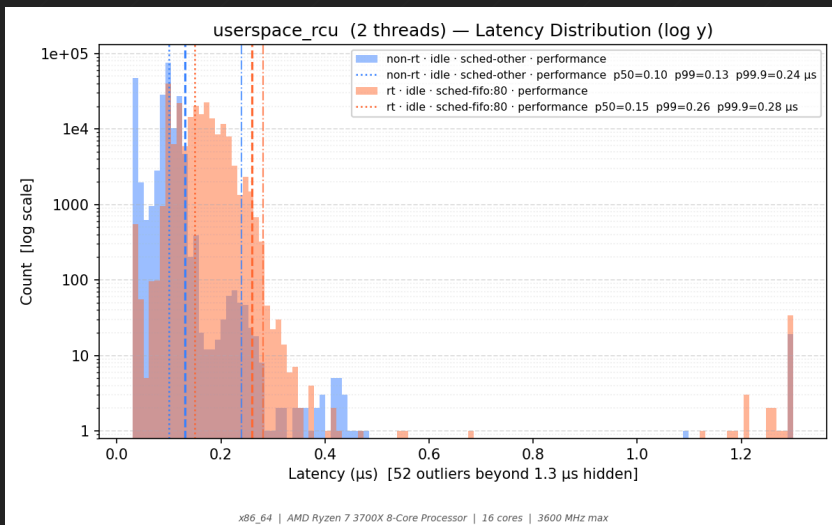
while (state.gen.load(
    memory_order_acquire) != expected)
    ; // no cpu_relax() — bare spin, monopolizes CPU under SCHED_FIFO

clock_gettime(CLOCK_MONOTONIC, &t_wake);
lat = diff_ns(state.signal_time, t_wake);
rcu_read_unlock();
sem_post(&state.ack);
```

userspace_rcu — x64 vs aarch64

x64

aarch64



△ On aarch64, userspace_rcu measures faster than busy_wait_atomic — benchmark artifact: UrcuState packs gen and signal_time in one cache

Entering the Kernel Costs 30–70×

SPINNING — OUR MEASUREMENTS

70–190 ns

x64: 70–130 ns · aarch64: 170–190 ns
Cache-coherency only — no kernel involved.

SLEEPING — OUR MEASUREMENTS

3.7–13 μs

x64: 3.7–4.1 μs · aarch64 RT: ~5 μs ·
aarch64: 10–13 μs
Kernel entry + scheduler overhead.

On x64 the gap was **~50×**. On aarch64 it ranged from 30–60× depending on kernel and load.

The exact numbers vary by platform and scheduler policy — but the order of magnitude holds. That's what we'll dig into.

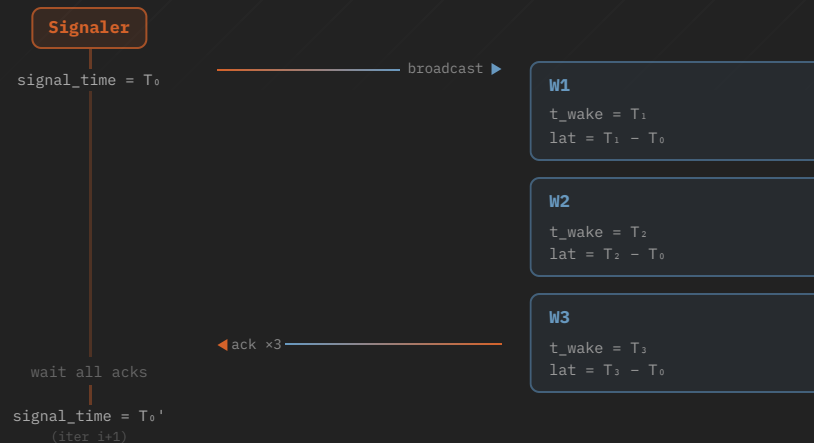
Multi-Waiter: What We Measure

N threads = 1 signaler + N-1 waiters.

Each iteration the signaler records `signal_time`, broadcasts once, and every waiter records its own `t_wake` independently — no ordering constraint between waiters.

Each waiter contributes **one latency sample per iteration**. All N-1 samples are merged into `LatencyStats` in iteration order.

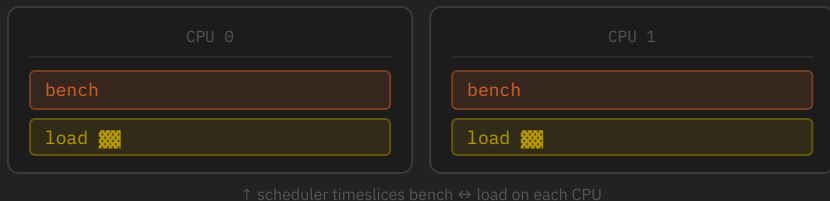
The signaler waits for **all N-1 acks** before starting the next iteration — the slowest waiter determines throughput and dominates the tail.



Load Simulation

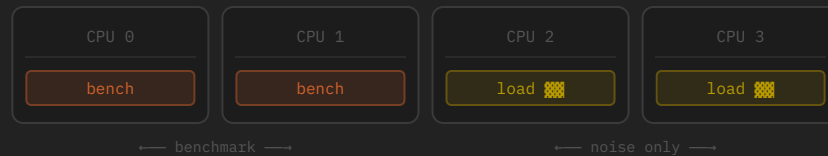
Two load threads spinning at 80% duty cycle + POSIX timer firing `SIGUSR1` at 1000 Hz — simulating CPU pressure and interrupt delivery. The affinity strategy determines what kind of stress the benchmark sees.

SAME CPUS — DIRECT SCHEDULER CONTENTION



Benchmark threads compete for CPU time with load threads. Wakeup latency is now bounded by the CFS timeslice (~4 ms), not the scheduler wakeup path.

DIFFERENT CPUS — BACKGROUND NOISE ONLY



Load is isolated to separate CPUs. Benchmark threads are never preempted — only cross-CPU cache coherency and interconnect pressure remains.

`std::condition_variable`

The classic C++ synchronization primitive

How `condvar` Works

A **condition variable** pairs with a **mutex**.

The waiter holds the lock, then calls `cv.wait(lk)`, which:

- **atomically enqueues on the wait queue before releasing the lock** — the kernel closes the lost-wakeup window: a notify cannot be missed once the thread is enqueued
- re-acquires the lock before returning

The signaler acquires the lock, updates shared state, releases it, then calls `cv.notify_all()`.

Correctness note: latency is computed *inside the waiter's lock* — preventing the signaler from overwriting `signal_time` before all waiters have read it.

SIGNALER

```
{ lock_guard lk(mtx);
  clock_gettime(CLOCK_MONOTONIC,
               &state.signal_time);
  state.acks_remaining = n_waiters;
  state.gen = i + 1; }
cv.notify_all();
```

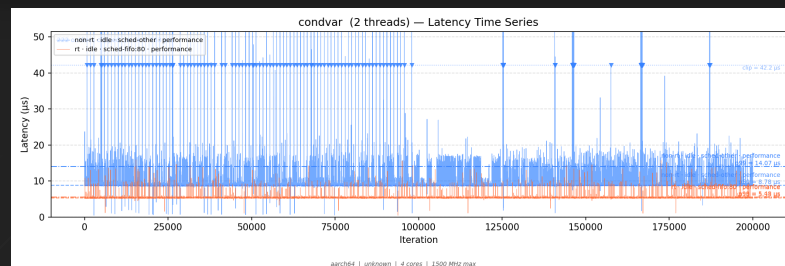
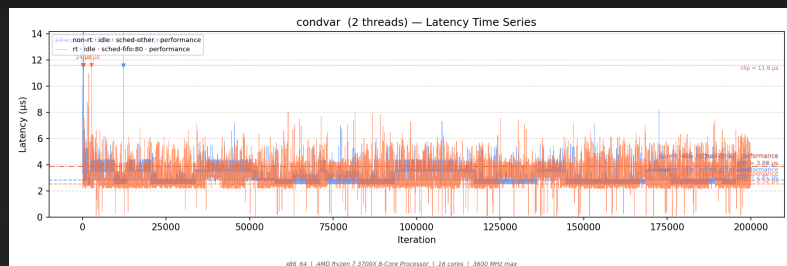
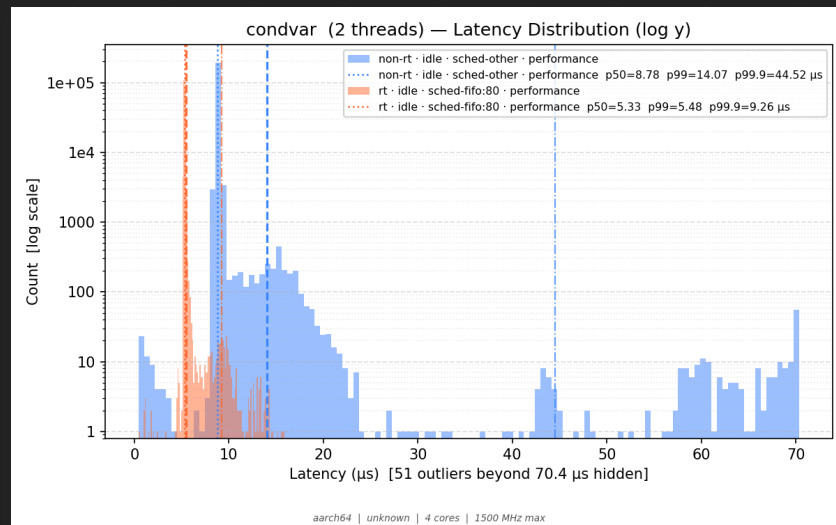
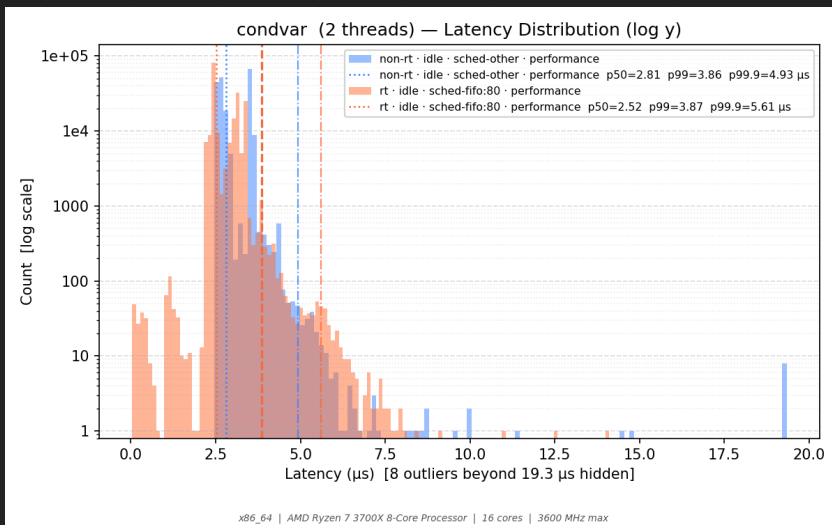
WAITER (PER THREAD W)

```
unique_lock lk(mtx);
while (state.gen <= last_gen) // spurious-wakeup guard
  cv.wait(lk); // atomically: unlock → sleep → re-lock
clock_gettime(CLOCK_MONOTONIC, &t_wake);
// diff inside lock - signal_time stable
lat = diff_ns(state.signal_time, t_wake);
--state.acks_remaining;
```

condvar — x64 vs aarch64

x64

aarch64



2t · idle · 1 M samples · histogram (main) · timeseries (spike pattern)

condvar — Multi-waiter Latency

Threads	x64	x64 RT	aarch64	aarch64 RT
2t	3.6 µs	3.9 µs	12.9 µs	5.5 µs
4t	5.2 µs	7.8 µs	14.1 µs	10.9 µs
8t	12.3 µs	25.2 µs	—	—
16t	22.9 µs	63.6 µs	—	—

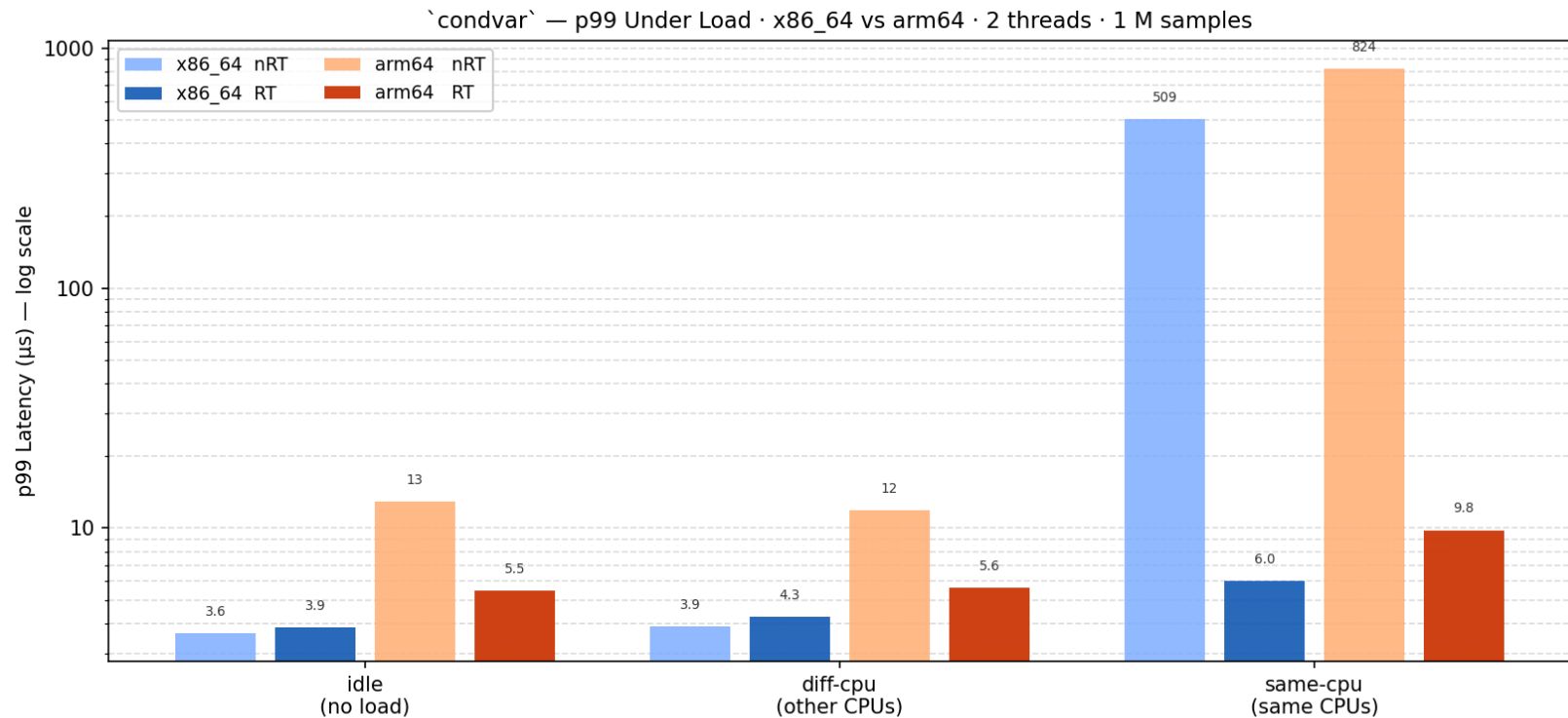
p99 · idle · 1 M samples per thread count

All N-1 waiters wake simultaneously from `notify_all` and immediately re-contend for the **same mutex**. Each waiter serializes through the lock before recording its timestamp — the last one in pays the full queuing cost.

Under SCHED_FIFO:80 this serialization is worse: RT threads don't yield voluntarily, so mutex hand-off latency compounds at each step. At 16t the last waiter waits behind 15 RT-priority mutex acquisitions.

At 8t, RT p99 (25 µs) is already 2× worse than the baseline (12 µs). The mutex is the bottleneck, not the kernel wakeup path.

condvar — Under Load



x86_64: AMD Ryzen 7 3700X · 16 cores · 3600 MHz | arm64: Cortex-A72 · 4 cores · 1500 MHz

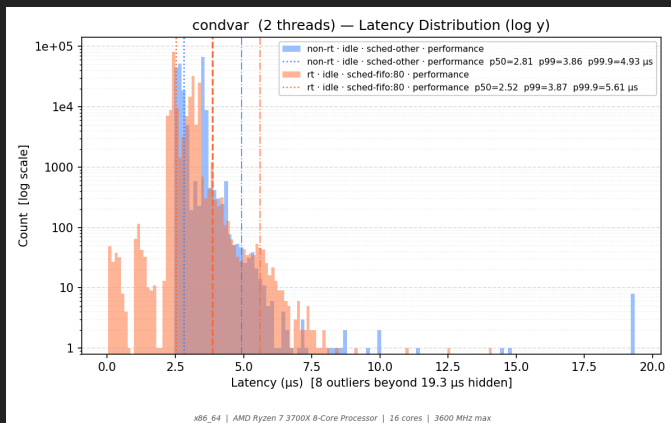
same-cpu: both architectures collapse (~500–820 μs) · x64 RT holds flat at 6 μs · aarch64 RT: partial shielding (10 μs same-cpu vs 5.5 μs)

pthread_condvar

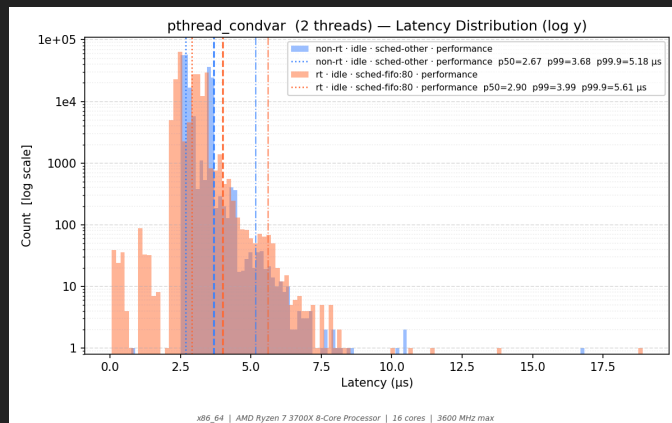
The POSIX primitive beneath `std::condition_variable` — verifying the zero-cost abstraction claim

condvar vs pthread_condvar — Is the C++ wrapper free?

`STD::CONDITION_VARIABLE` (C++)



`PTHREAD_COND_BROADCAST` (POSIX DIRECT)



Threads	x64	x64 RT	aarch64	aarch64 RT	Threads	x64	x64 RT	aarch64	aarch64 RT
2t	3.6 μs	3.9 μs	12.9 μs	5.5 μs	2t	3.9 μs	3.9 μs	12.2 μs	5.4 μs
4t	5.2 μs	7.8 μs	14.1 μs	10.9 μs	4t	5.7 μs	7.8 μs	13.6 μs	9.9 μs
16t	22.9 μs	63.6 μs	—	—	16t	24.7 μs	63.6 μs	—	—

eventfd

A Linux file descriptor as a signaling channel

How `eventfd` Works

`eventfd(2)` — a kernel-managed **64-bit counter** behind a fd.

- `write(fd, &one, 8)` → adds 1, wakes any blocked reader
- `read(fd, &val, 8)` → blocks until counter > 0, resets it

No mutex needed. `write()` is the happens-before edge: the waiter cannot return from `read()` until `signal_time` is visible.

`read()` adds the thread to the wait queue, then calls `schedule()`. `write()` calls `wake_up()` on that queue — the same queue `epoll_wait` polls, which is why `eventfd` integrates natively with I/O multiplexing.

SIGNALER

```
uint64_t val;
for (w = 0; w < n_waiters; ++w)
    read(fd_ack.fd, &val, 8);    // block until each ack

clock_gettime(CLOCK_MONOTONIC, &state.signal_time);
uint64_t one = 1;
for (w = 0; w < n_waiters; ++w)
    write(fd_sigs[w].fd, &one, 8);    // wake waiter w
```

WAITER (PER THREAD W)

```
uint64_t val;
read(fd_sigs[w].fd, &val, 8);    // block until write() arrives

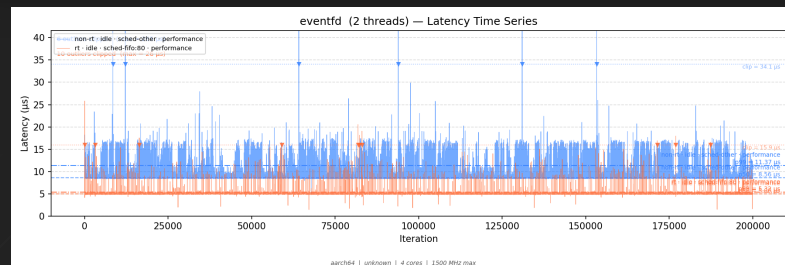
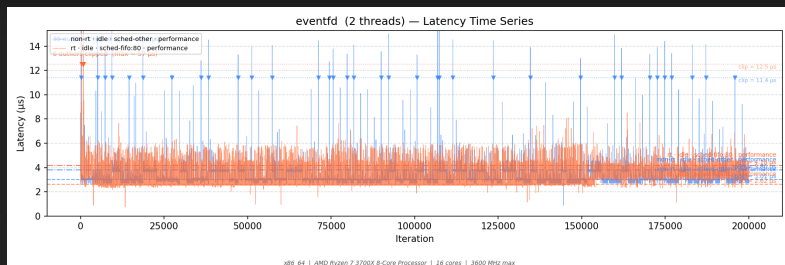
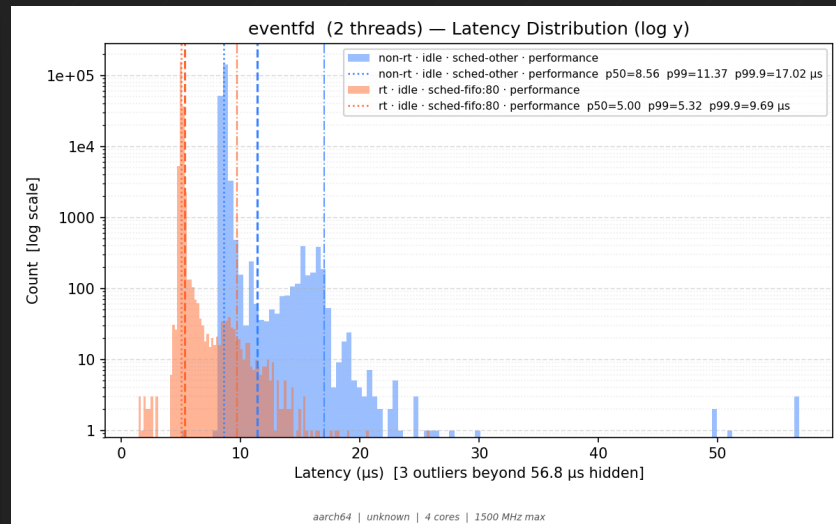
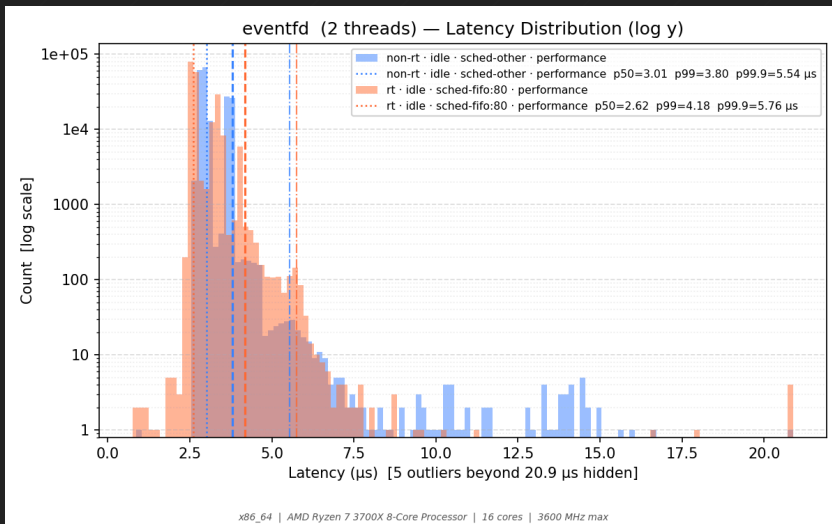
clock_gettime(CLOCK_MONOTONIC, &t_wake);
lat = diff_ns(state.signal_time, t_wake);

uint64_t one = 1;
write(fd_ack.fd, &one, 8);    // ack signaler
```

eventfd — x64 vs aarch64

x64

aarch64



2t · idle · 1 M samples · histogram (main) · timeseries (spike pattern)

eventfd — Multi-waiter Latency

Threads	x64	x64 RT	aarch64	aarch64 RT
2t	3.8 µs	4.1 µs	11.4 µs	5.3 µs
4t	5.6 µs	5.7 µs	14.1 µs	10.7 µs
8t	10.1 µs	13.5 µs	—	—
16t	16.7 µs	27.2 µs	—	—

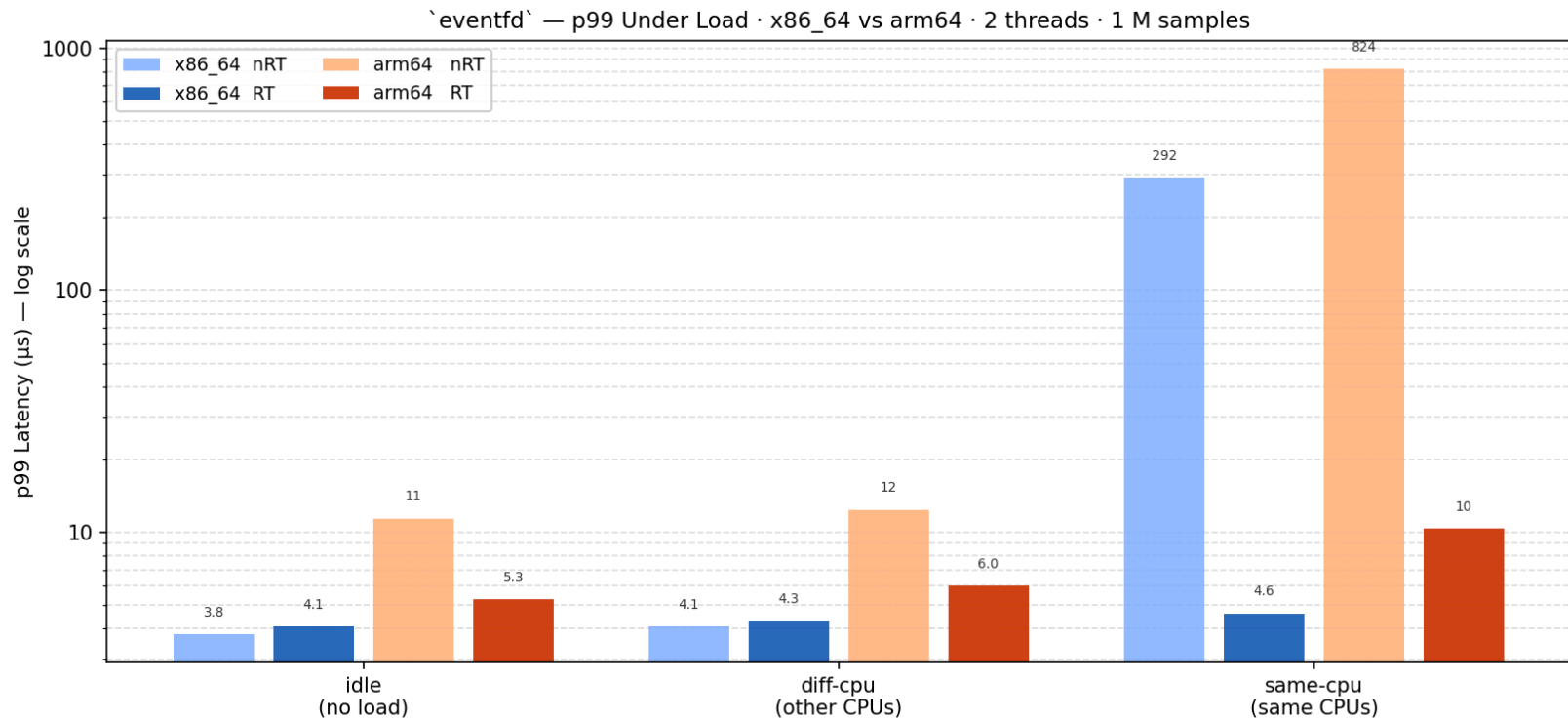
p99 · idle · 1 M samples per thread count

Each waiter has its own dedicated `fd_sig` — the signaler writes to each independently. **No shared lock or broadcast contention between waiters.** At 4t, RT and the baseline are nearly identical (5.7 vs 5.6 µs).

Scales more gracefully than condvar/futex under RT: 27 µs at 16t vs 64–71 µs for broadcast primitives.

Best RT scaling among sleeping primitives at 16t, alongside `posix_sem`. The per-waiter fd design eliminates the wakeup contention point.

eventfd — Under Load



x86_64: AMD Ryzen 7 3700X · 16 cores · 3600 MHz | arm64: Cortex-A72 · 4 cores · 1500 MHz

fd syscall floor (~670 ns) visible as the minimum in every condition · same-cpu: both arches collapse · x64 RT flat at 6 μs · aarch64 RT: 10

posix_sem

The POSIX standard: `sem_post` / `sem_wait`

How `posix_sem` Works

`sem_init(2)` — a **POSIX semaphore**: a kernel counter threads can post and wait on.

- `sem_post(&s)` → increments counter, wakes one blocked waiter
- `sem_wait(&s)` → decrements counter if > 0, else blocks

No mutex on the wakeup path. `sem_post` is the happens-before edge — `signal_time` is visible as soon as `sem_wait` returns.

Sleep path: `sem_wait()` wraps `futex` (glibc): fast path if counter > 0 — no syscall; otherwise `futex(FUTEX_WAIT)`.
In strict ping-pong the counter is always 0 on arrival, so every iteration blocks.

SIGNALER

```
for (w = 0; w < n_waiters; ++w)
    sem_wait(&sem_ack);    // block until each ack

clock_gettime(CLOCK_MONOTONIC, &state.signal_time);
for (w = 0; w < n_waiters; ++w)
    sem_post(&sem_sigs[w]); // wake waiter w
```

WAITER (PER THREAD W)

```
sem_wait(&sem_sigs[w]);    // blocks until signaler posts

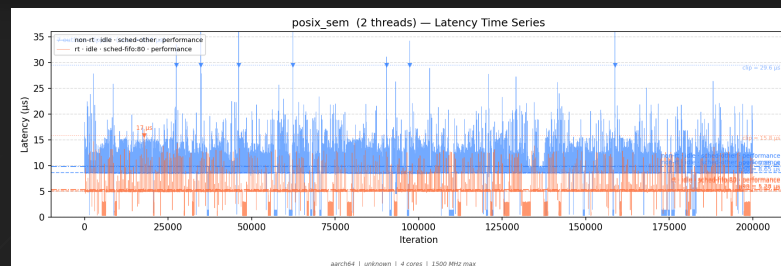
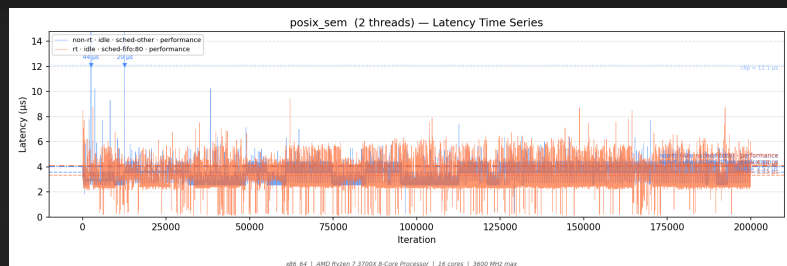
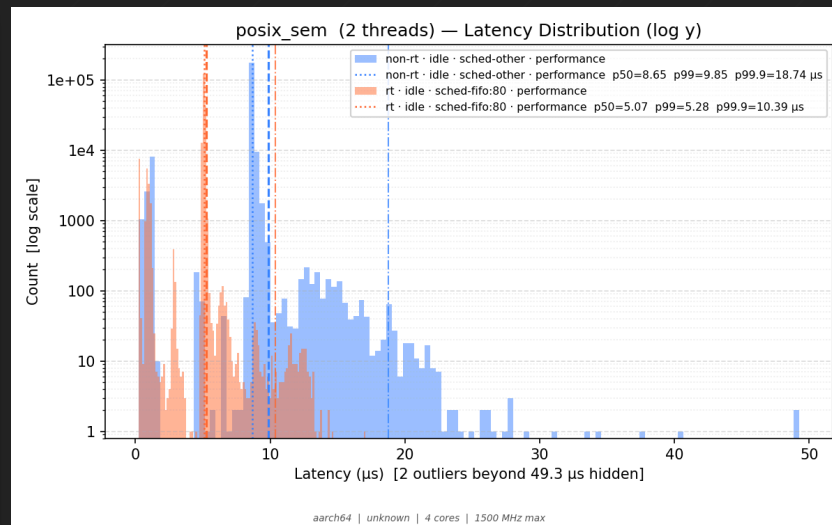
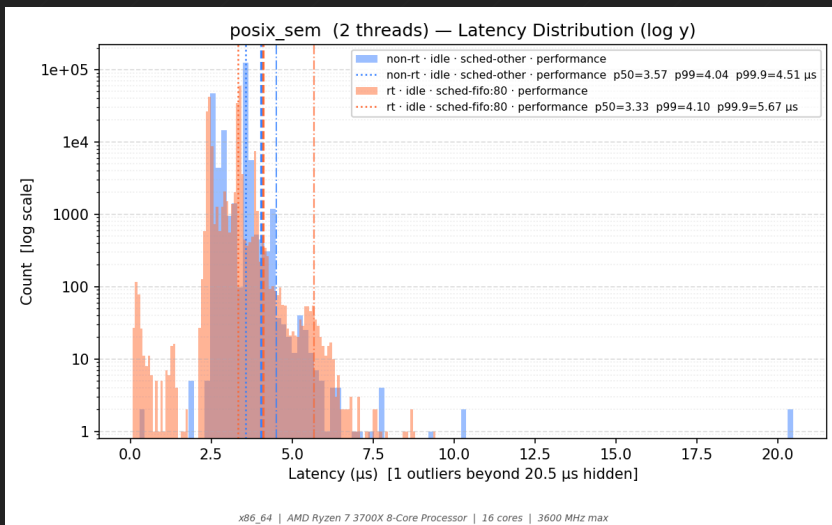
clock_gettime(CLOCK_MONOTONIC, &t_wake);
lat = diff_ns(state.signal_time, t_wake);

sem_post(&sem_ack);        // ack signaler
```

posix_sem — x64 vs aarch64

x64

aarch64



2t · idle · 1 M samples · histogram (main) · timeseries (spike pattern)

posix_sem — Multi-waiter Latency

Threads	x64	x64 RT	aarch64	aarch64 RT
2t	3.7 μ s	4.0 μ s	9.9 μ s	5.3 μ s
4t	5.6 μ s	14.2 μ s	16.5 μ s	13.1 μ s
8t	9.5 μ s	24.0 μ s	—	—
16t	15.8 μ s	26.2 μ s	—	—

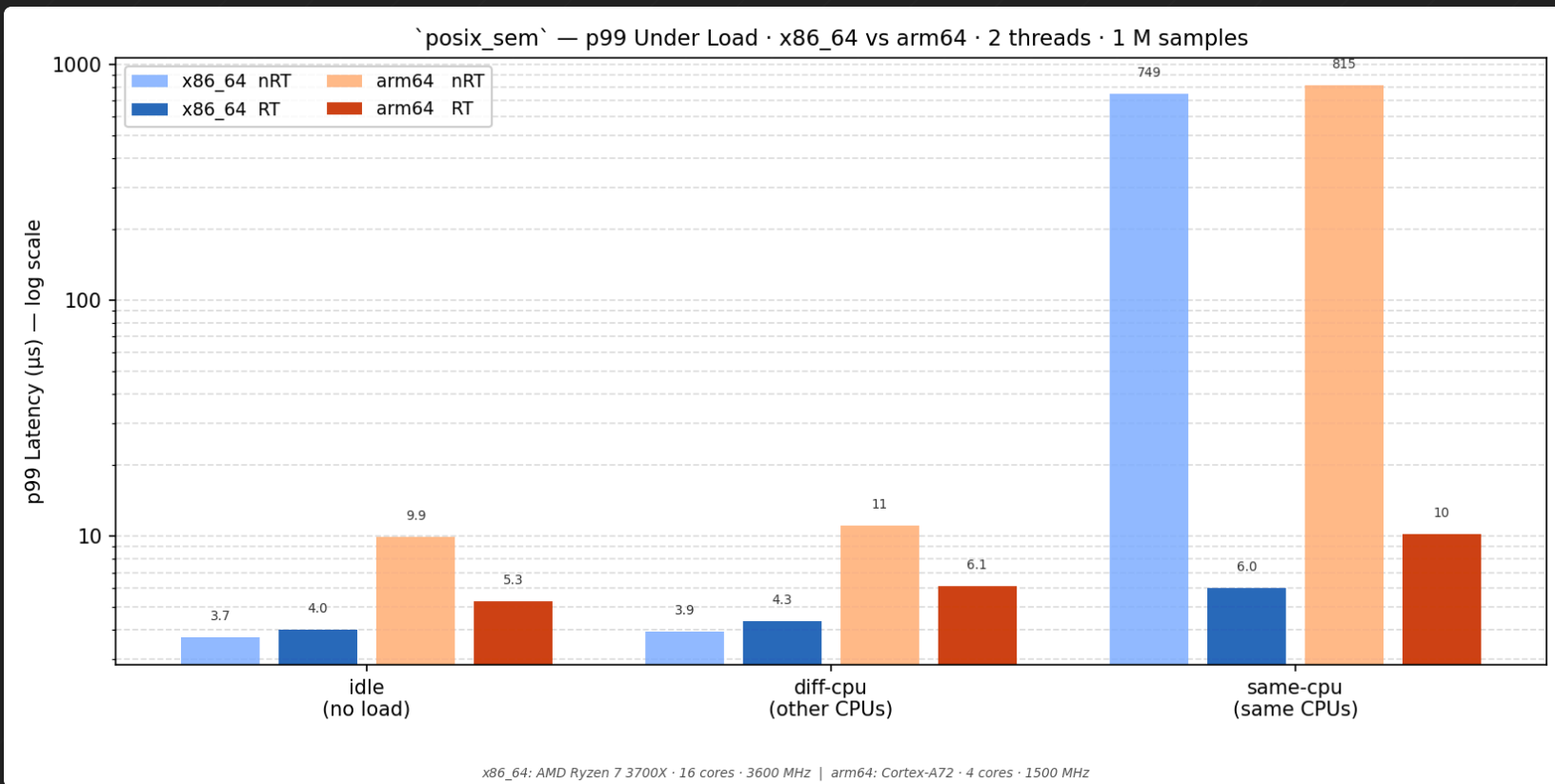
p99 · idle · 1 M samples per thread count

Scales cleanly — best at 16t among all sleeping primitives (15.8 μ s). Per-waiter semaphores mean no shared wakeup contention, same as `eventfd`.

RT shows a sharp jump at 4t (4.0 $\rightarrow$ 14.2 μ s) then levels off. At 16t RT p99 (26.2 μ s) is nearly identical to `eventfd` (27.2 μ s). The `sem_post` path under SCHED_FIFO appears to serialize differently than the fd write path at low counts.

For fan-out, `posix_sem` is the best-scaling sleeping primitive across the 2–16t range.

posix_sem — Under Load



aarch64 RT shows a sharp 4t jump (4 → 14 μs) then levels off — `sem_post` serialization under `SCHED_FIFO` differs from the `fd` write path ·

futex

The raw syscall beneath everything else

How `futex` Works

`futex` is the low-level primitive a lot of library primitives are built on. We can use it directly:

- `FUTEX_WAIT(addr, val)` — sleep if `*addr == val` (kernel checks atomically)
- `FUTEX_WAKE(addr, n)` — wake up to N waiters on `addr`

Two `futex` words: `gen` (generation counter, for broadcast) and `ack` (countdown, for fan-in).

Fast path: the waiter checks `gen` in user space before calling into the kernel. If the signaler has already advanced it, no syscall at all.

Broadcast: all N-1 waiters sleep on the same `gen` address — one `FUTEX_WAKE(INT_MAX)` wakes them all simultaneously.

SIGNALER

```
while (ack.load(acquire) != 0) // wait for all prev acks
    futex_wait(&ack, ack.load());

ack.store(n_waiters, relaxed);
clock_gettime(CLOCK_MONOTONIC, &state.signal_time);
gen.fetch_add(1, release);      // signal_time now visible
futex_wake(&gen, INT_MAX);      // wake all N waiters at once
```

WAITER (PER THREAD W)

```
int expected = last_gen;
// fast path: if gen already advanced, skip this block entirely
if (gen.load(acquire) == expected) // still old → must sleep
    futex_wait(&gen, expected);     // kernel re-checks atomically
last_gen = gen.load(acquire);

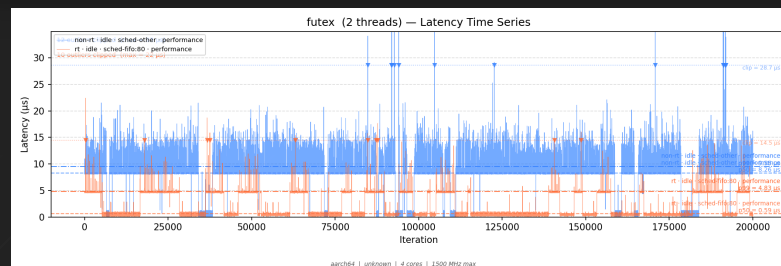
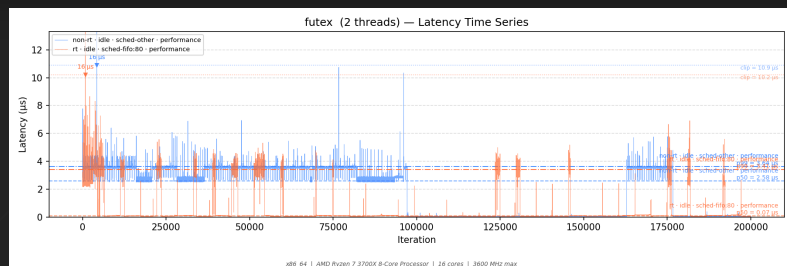
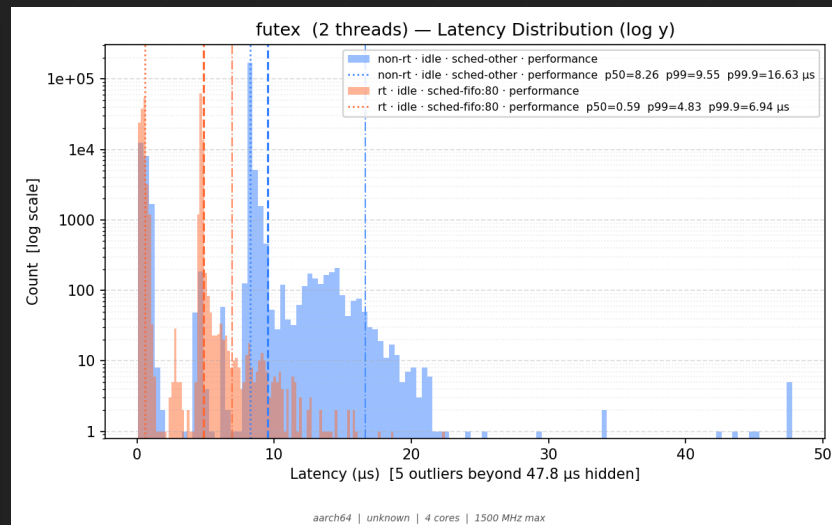
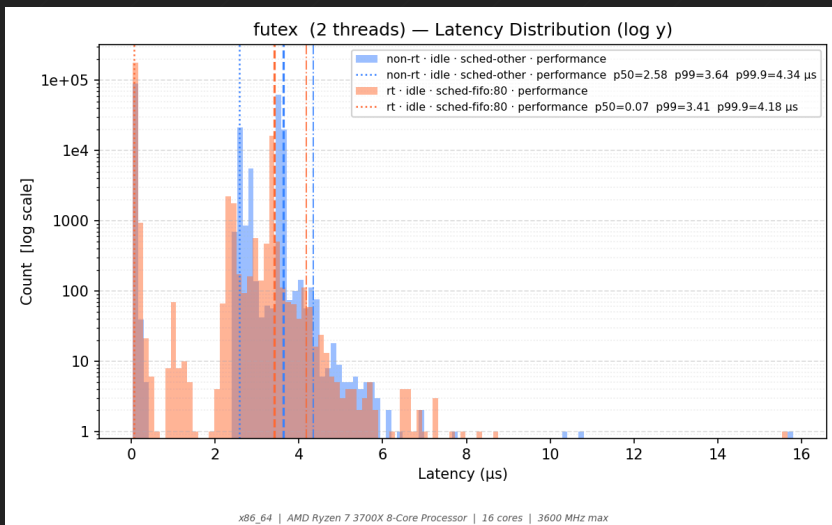
clock_gettime(CLOCK_MONOTONIC, &t_wake);
lat = diff_ns(state.signal_time, t_wake);

if (ack.fetch_sub(1, release) == 1)
    futex_wake(&ack, 1);           // last ack: wake signaler
```

futex — x64 vs aarch64

x64

aarch64



aarch64 RT: bimodal — fast-path at p50, kernel path at p99

futex — Multi-waiter Latency

Threads	x64	x64 RT	aarch64	aarch64 RT
2t	3.6 µs	3.7 µs	9.7 µs	4.8 µs
4t	4.7 µs	12.4 µs	13.4 µs	8.4 µs
8t	7.1 µs	27.5 µs	—	—
16t	13.3 µs	71.1 µs	—	—

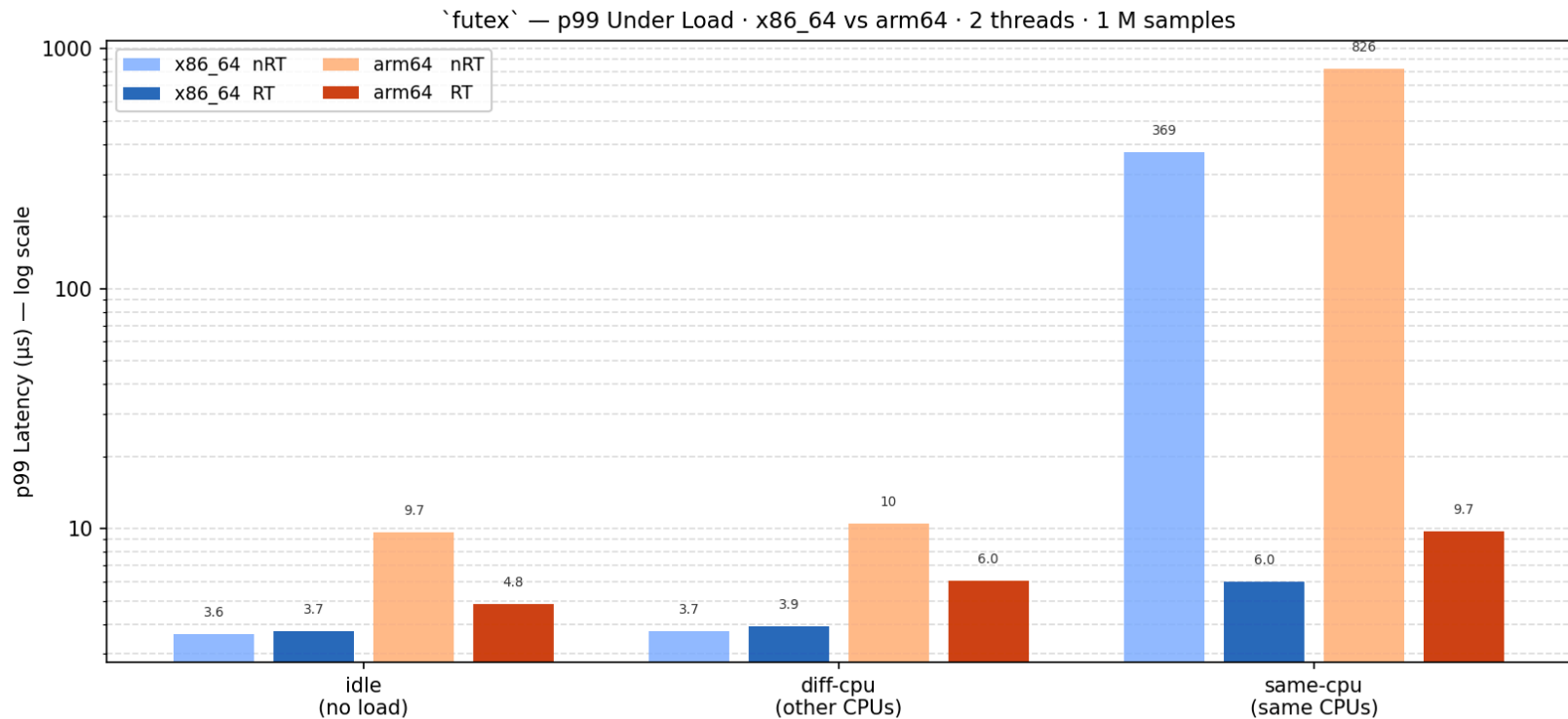
p99 · idle · 1 M samples per thread count

The sharpest inversion in the dataset. Without RT it's the best-scaling (3.6 → 13.3 µs, 4×); under RT it's the worst (3.7 → 71.1 µs, 19×).

All N−1 waiters wake up simultaneously. Under `SCHED_OTHER`, CFS work-stealing aggressively distributes newly-runnable threads across idle CPUs — the waiters fan out in parallel. Under `SCHED_FIFO:80` this changes fundamentally: CFS load-balancing is explicitly disabled for RT tasks. Equal-priority `SCHED_FIFO` threads don't preempt each other and run FIFO — so the waiters execute more sequentially.

Do not use futex for high fan-out under RT. At 16t its RT p99 (71 µs) is 3× worse than `posix_sem` or `eventfd` (26–27 µs).

futex — Under Load



x86_64: AMD Ryzen 7 3700X · 16 cores · 3600 MHz | arm64: Cortex-A72 · 4 cores · 1500 MHz

same-cpu: both arches collapse · x64 RT flat · aarch64 RT: partial shielding (10 μ s same-cpu vs 5 μ s idle) · aarch64 RT idle is low due to fast-path dominance

Two Signaling Models

You've now seen the per-waiter tables for all five primitives. The signal model explains why some scale while others don't.

BROADCAST — CONDVAR, FUTEX

One call wakes all N-1 waiters simultaneously from a shared address.

```
// condvar
cv.notify_all();

// futex
FUTEX_WAKE_PRIVATE(&gen, INT_MAX)
```

Under **SCHED_OTHER**: CFS distributes waiters fairly — scales OK.

Under **SCHED_FIFO:80**: equal-priority run-queue insertions serialize —

`futex` degrades **19× at 16t** (3.7 → 71 μs).

PER-WAITER — EVENTFD, POSIX_SEM

Signaler loops and posts to each waiter's dedicated channel.

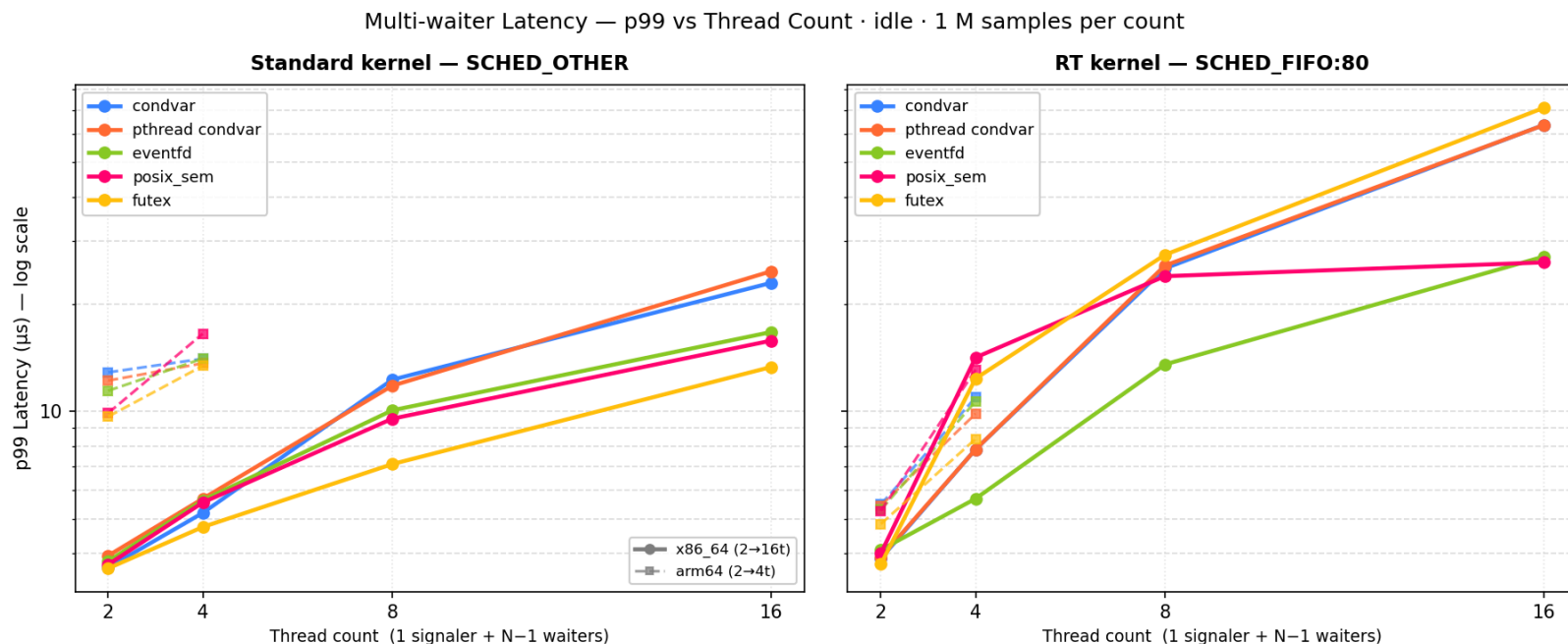
```
// eventfd / posix_sem
for (w = 0; w < n_waiters; ++w)
    write(fd_sigs[w].fd, &one, 8);
// or: sem_post(&sem_sigs[w])
```

No shared contention at wakeup. Each waiter owns its signal channel — scales **7× at 16t** under RT vs 19× for broadcast.

Multi-waiter Latency

1 signaler · 2 → 4 → 8 → 16 waiters · sleeping primitives · 1 M samples per thread count

Multi-waiter Latency: p99 vs Thread Count



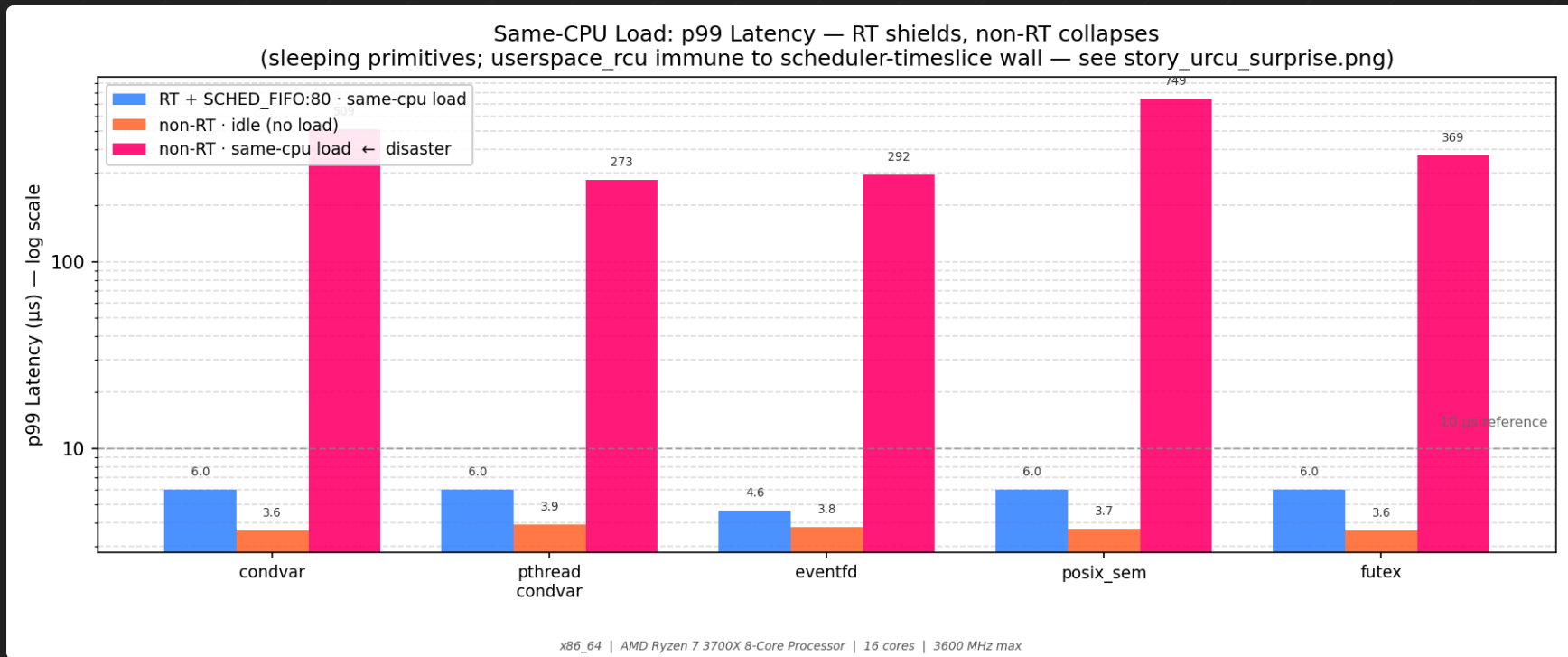
x86_64: AMD Ryzen 7 3700X · 16 cores · 3600 MHz | arm64: Cortex-A72 · 4 cores · 1500 MHz

At 2t all primitives are within 1 μ s of each other. Without RT, `futex` scales best (13 μ s at 16t). Under RT, `futex` becomes the **worst** (71 μ s at 16t). `eventfd` and `posix_sem` (per-waiter signaling) hold flat across both policies: **the right choice for RT fan-out**.

What Load Does to Latency

80% CPU duty · 1000 IRQ/s · same-cpu vs diff-cpu · SCHED_OTHER and SCHED_FIFO:80

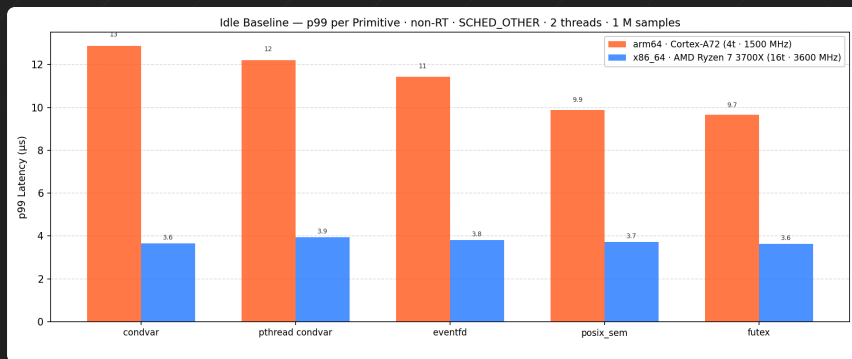
Without RT: Same-CPU Collapses p99



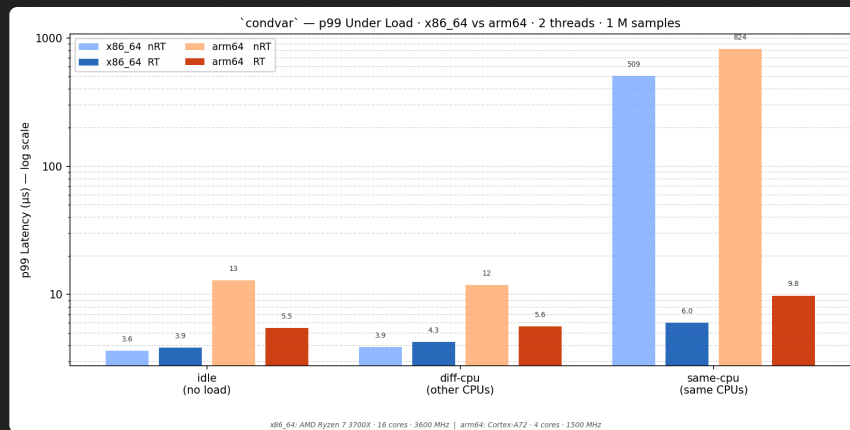
same-CPU load: p99 jumps to **600–800 μs** (100–200× vs idle) across every sleeping primitive. diff-cpu stays within noise of idle.

Primitive choice becomes irrelevant — the scheduler determines everything.

aarch64 (Cortex-A72) — Same Story, Different Scale



aarch64 idle p99 is **~3× higher** than x64 (10–13 μs vs 3.6–3.9 μs). Slightly more than the 2.4× clock difference — Cortex-A72 scheduler entry carries more overhead. Primitive ranking is consistent.

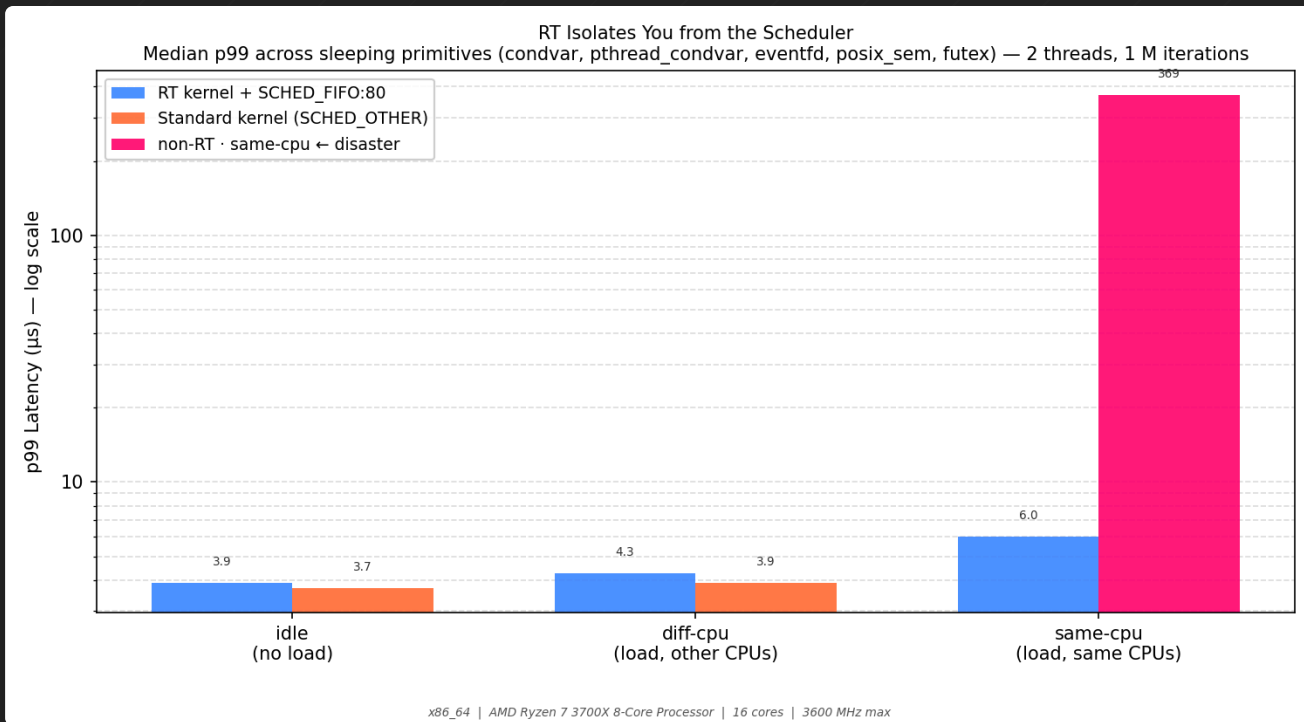


Same-CPU collapse hits both architectures — **509 μs (x64) · 824 μs (aarch64)** — despite the clock difference. The scheduler timeslice is wall-clock time. RT holds flat on both: 6.0 μs (x64) · 9.8 μs (aarch64).

With and Without PREEMPT_RT

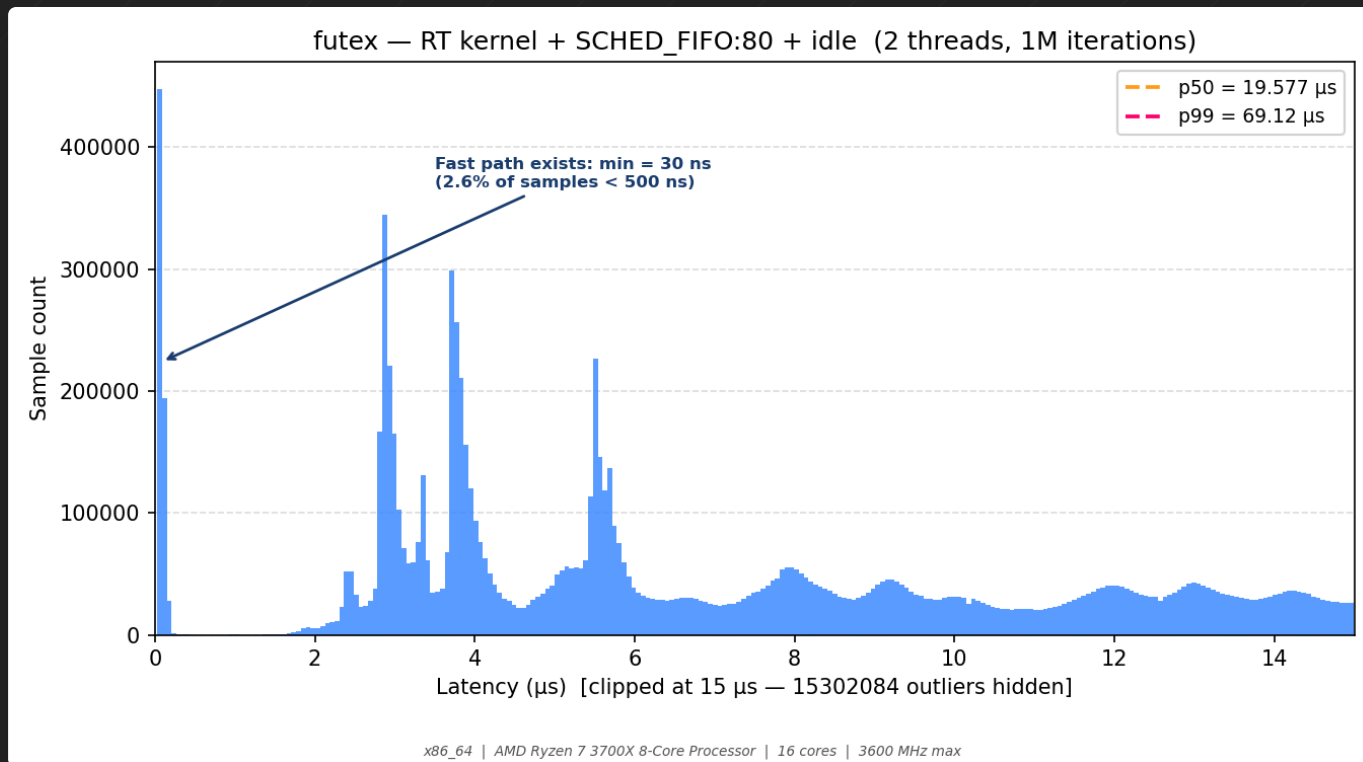
SCHED_FIFO:80 on Linux 6.6.87-rt54 · 6 run conditions · 1 M samples each

RT + SCHED_FIFO:80 — Similar Median, Capped Tail



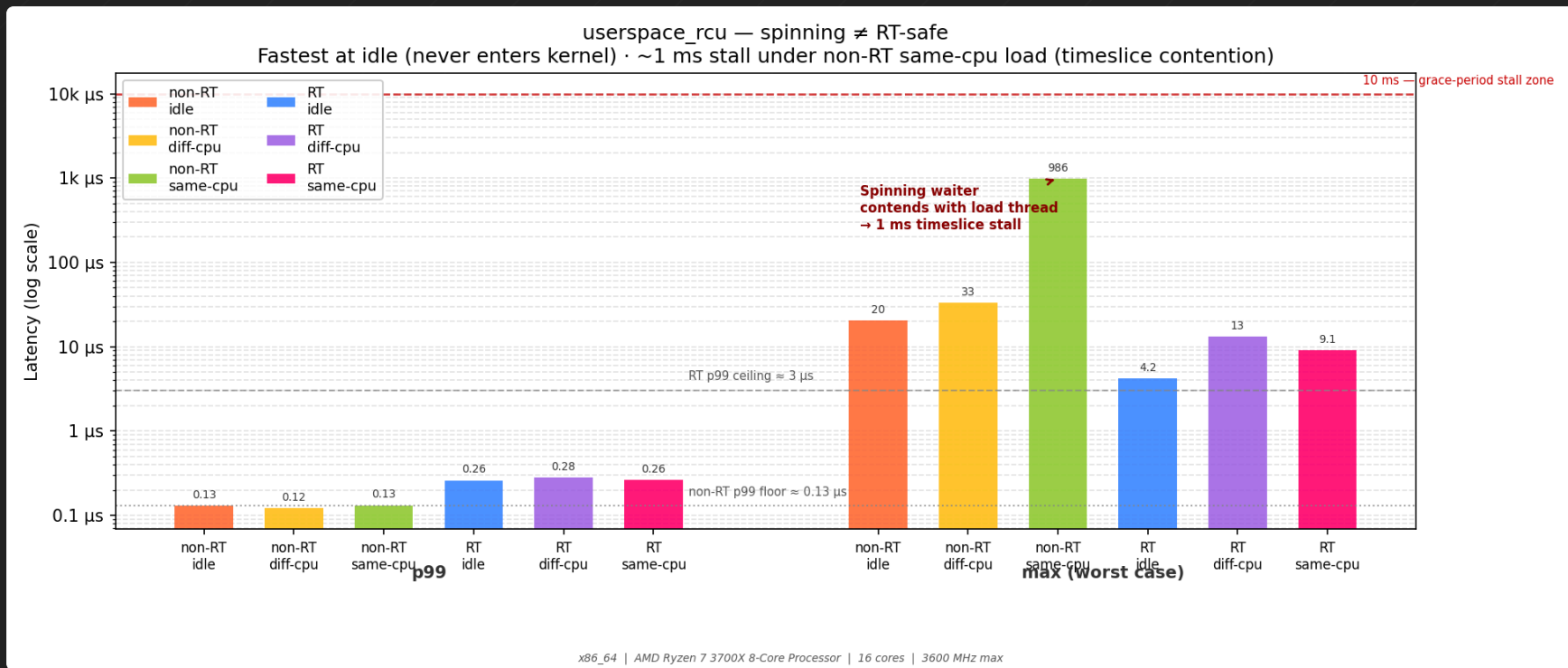
Idle and diff-cpu: RT and non-RT within noise of each other (~3.9 μs p99). Same-cpu load: non-RT collapses to **369 μs p99** · RT holds flat at **6 μs** — a 60× difference. Median p99 across sleeping primitives · log scale · userspace_rcu excluded.

The Futex Fast Path



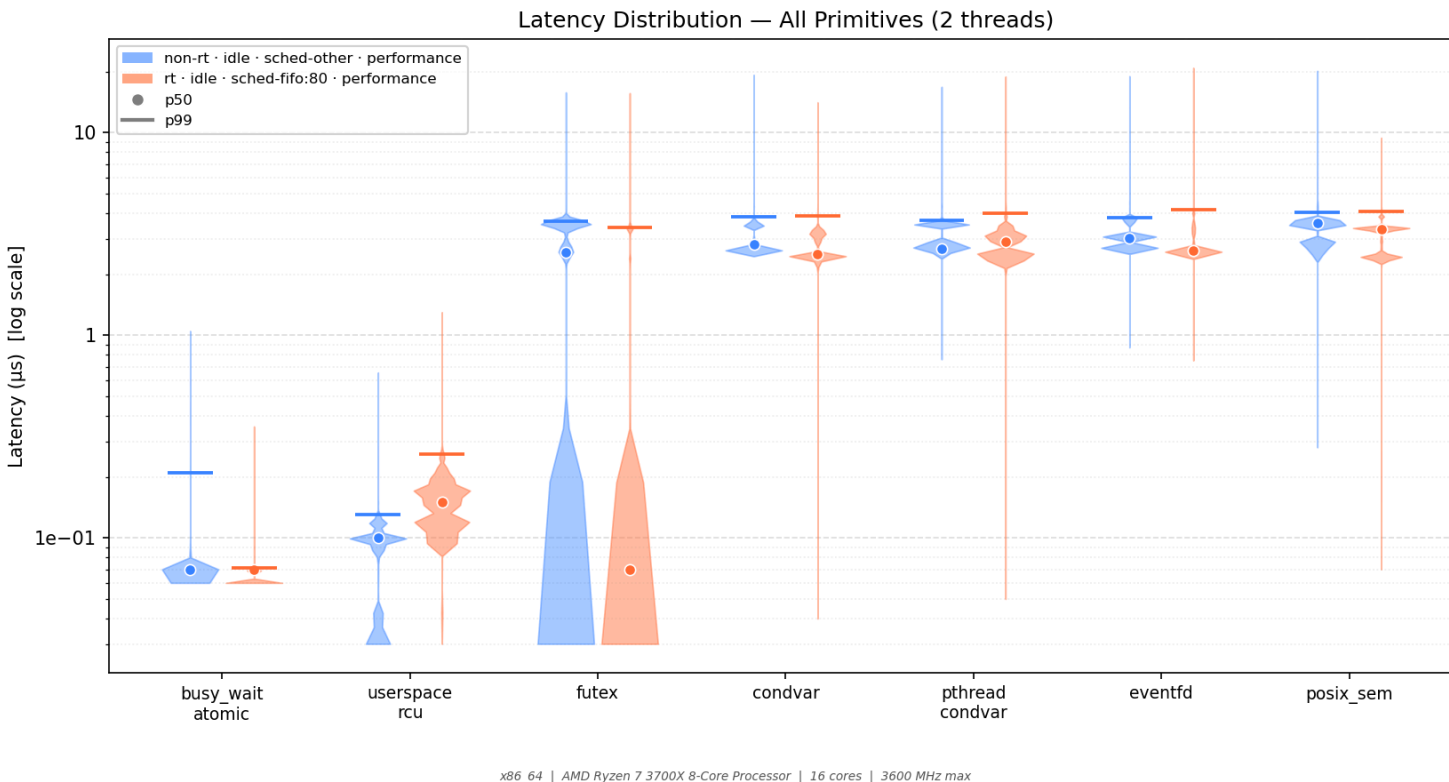
The futex user-space fast path exists: min = **30 ns**, confirming the generation counter can be observed without a syscall. In this run it's rare (p50 $\approx$ 2.5 μ s) but elevated stddev vs condvar signals it does occur. Only futex exposes this path — all other sleeping primitives have structural syscall floors.

The userspace_rcu Inversion



p99 = **130 ns** — 20× below sleeping primitives, because the waiter never enters the kernel. Non-RT same-cpu max = **986 μ s**: the spinning waiter gets timesliced against the load thread sharing its CPU — a ~1 ms stall. RT idle max = 4.2 μ s looks safe, but the risk is structural: under SCHED_FIFO:80 one preemption miss stalls the waiter for a full timeslice. **Lock-free at user level does not mean RT-safe.**

Head-to-Head Comparison



Near-identical rank order; RT tightens the tail on sleeping primitives without moving the median much.

The Rankings

Scenario	Best	Worst	Verdict
2t · idle · nRT	futex 3.6 µs	pthread_condvar 3.9 µs	0.31 µs spread — noise, no meaningful ranking
2t · idle · RT	futex 3.7 µs	eventfd 4.1 µs	0.36 µs spread — same story
16t · idle · nRT	futex 13.3 µs	pthread_condvar 24.7 µs	futex leads; mutex-based primitives lag
16t · idle · RT	posix_sem 26 µs · eventfd 27 µs	futex 71 µs	inversion — futex best→worst under RT fan-out
2t · same-cpu · nRT	—	all 270–750 µs	all collapse; scheduler timeslice dominates
2t · same-cpu · RT	eventfd 4.6 µs	others ~6 µs	all flat — RT policy is everything

AT 2T

Primitive choice is effectively irrelevant — 0.3 µs spread. Pick by API ergonomics.

AT 16T UNDER RT

futex WAKE(INT_MAX) serializes equal-priority RT threads — 19× degradation. posix_sem / eventfd (per-waiter) hold flat at 7×.

⚠ SPINNING ≠ RT-SAFE

userspace_rcu p99 = 130 ns but never yields the CPU. Under SCHED_FIFO:80 one missed preemption may stall waiter a full timeslice.

✓ DIFF-CPU LOAD IS BENIGN

p99 stays at 3.7–4.3 µs under diff-cpu load — nearly identical to idle. Cross-CPU coherency is not the dominant factor. **Affinity before RT.**

The Rubric

Need	Standard kernel	RT kernel (SCHED_FIFO)	Notes
Lowest p99, idle	x64: any 3.6–3.9 μ s · aarch64: <code>futex</code> / <code>posix_sem</code> 9.7 μ s, <code>condvar</code> 12.9 μ s	x64: any 3.7–4.1 μ s · aarch64: <code>futex</code> 4.8 μ s, others 5.3–5.5 μ s	<code>futex</code> fast path: x64 p50=0.1 μ s, aarch64 p50=0.6 μ s
Under same-CPU load	x64 p99: 272–748 μ s · aarch64: 815–826 μ s — all collapse	x64: ≤ 6 μ s p99 · aarch64: ≤ 11 μ s — all hold flat	Scheduler policy dominates. Primitive choice irrelevant.
Lowest max, idle	x64: <code>futex</code> 16 μ s · aarch64: <code>futex</code> 151 μ s	x64: <code>posix_sem</code> 12 μ s · aarch64: <code>condvar</code> 17 μ s	aarch64 non-RT idle max 151–550 μ s — OS jitter even at idle
Fan-out RT (N > 2t)	<code>futex</code> scales best (x64 13 μ s at 16t)	<code>posix_sem</code> / <code>eventfd</code> (26–27 μ s at 16t) vs <code>futex</code> (71 μ s)	<code>futex</code> WAKE(INT_MAX) serializes equal-priority RT threads
Portability + ecosystem	<code>condvar</code> , <code>posix_sem</code>	<code>condvar</code> , <code>posix_sem</code>	POSIX standard, any platform

x64: AMD Ryzen 7 3700X · Linux 6.6.135 / 6.6.87-rt54 · aarch64: Cortex-A72 · Linux 7.0.3 / 6.18.29-v8-rt+ · 1 M samples · 2 threads · idle unless noted.

Real-Time Linux in Real life !

Simulink Real-Time

Deploy directly to real-time targets. The same synchronization primitives benchmarked here underpin the runtime scheduler.

Supports Rapid Control Prototyping (RCP) and Hardware-in-the-Loop (HIL) testing. Multi-threaded real-time applications with μ s execution on CPU and ns on FPGA.

mathworks.com/products/simulink-real-time

Speedgoat Target Machines

Purpose-built hardware for Simulink Real-Time.

Runs a tuned real-time Linux RTOS — the stable, high-performance foundation that makes deterministic thread wakeup a product guarantee.

speedgoat.com

Key Takeaways

- ✓ Measure **p99, p99.9, and max** — not median. At idle, p50 is similar with and without RT. Max is where they diverge by 100×
- ✗ Same-CPU load without RT: p99 collapses to **270–750 µs** · **max ~800 µs** — the CFS timeslice becomes the ceiling. Idle benchmarks cannot predict this.
- ✓ RT + SCHED_FIFO:80 holds **p99 ≤ 6 µs** under the same load. **Kernel + scheduler policy dominates primitive choice.**
- ≈ At 2 threads idle, all sleeping primitives are within **0.31 µs** of each other — noise. Pick by API ergonomics, not benchmarks.
- 📈 Fan-out under RT inverts the ranking: `futex` degrades **19× at 16t** (3.7 → 71 µs); `posix_sem` / `eventfd` only **7×** (4 → 26 µs). Per-waiter signaling scales.

