

pg_wait_tracer

Beyond Sampling

True wait event tracing for PostgreSQL

Dmitry Fomin

Real PostgreSQL, a real disk, plain SQL workloads.

Five cases where the sampler points the wrong way

1

Short-request tail

The profile ranks the wrong fix

2

Lock convoy

0% locks for the query it hurts

3

Commit convoy

Blames locks, not the disk

4

Slow disk or bloat

Two causes, one profile

5

CPU limit

The sampler is throttled too

Sampler: pg_wait_sampling, every 10 ms, the usual dashboard panel

Trace: pg_wait_tracer (eBPF, no patch) and the wait-event timing patch (in core)

Measured October 2026 · Hetzner 4 vCPU AMD EPYC VMs · PostgreSQL 18.6 and 20devel + patch · pg_wait_sampling 1.1.11 at defaults · raw data and scripts in the pg_wait_tracer repo

At 100 Hz, a 5 ms wait is a coin flip



Share of injected lock waits that left any sample. No sample does not mean no wait.

Measured: 3,827 advisory-lock waits vs pg_wait_sampling history (100 Hz) · matches $\min(1, \text{duration} \div 10 \text{ ms})$ within 3 points, idle and under TPC-B

Case 1 of 5 · a ~1.5 ms checkout request at 2,000/s, with a 10–25 ms tail

The profile ranks the wrong fix

pg_wait_sampling: share of all wait time



reads 12% · **row locks 2.3%: ranked last**

Trace: wait time inside the slow requests (> 8 ms)



pg_wait_tracer 71% · the patch 60%: **same answer**

p99.9 latency after each fix

Baseline



synchronous_commit = off (profile's #1)



2 GB cache, no reads (profile's #3)



Restock job in 5-row chunks (the trace)

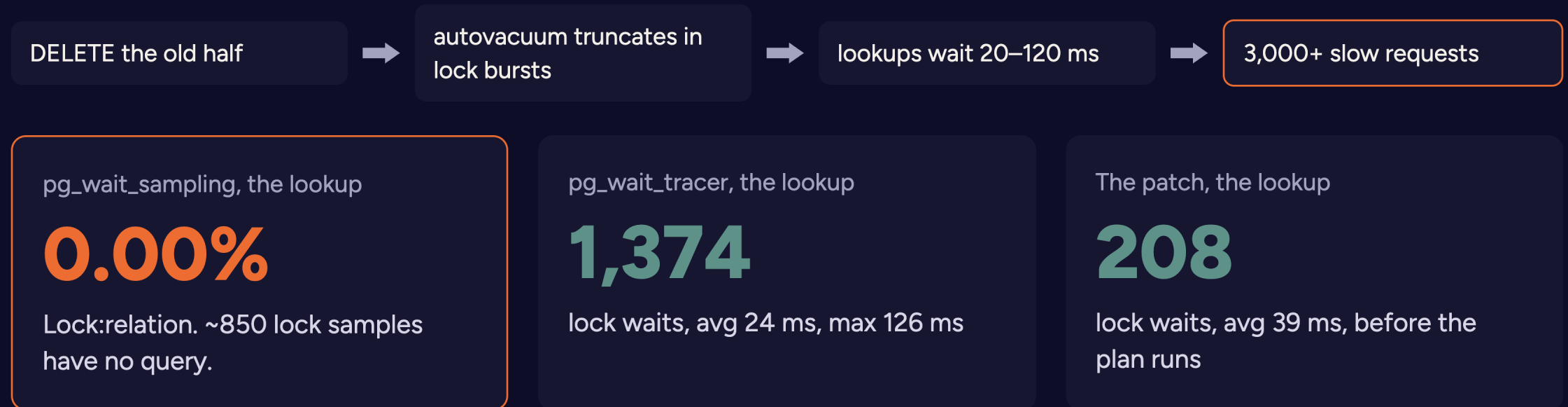


A background job holds 100 stock rows for ~17 ms. It is 2% of the time and most of the tail.

pgbench 16 clients, -M prepared, fixed 2,000 req/s, 120 s · BEGIN, catalog read, UPDATE stock, COMMIT · fixes on PostgreSQL 20devel + patch, trace also on PG18 + pg_wait_tracer · one run per fix, baseline twice; chunking repeated 3×

Case 2 of 5 · 8,000 primary-key lookups/s while autovacuum truncates the table

Lock convoy: no query in the sampler



The lock is taken at bind, before the query id is set. **vacuum_truncate = off: 1–10 slow requests.**

20 M rows, shared_buffers 2 GB · pgbench 16 clients, -M prepared · held in 3 sampler-only, 3 traced, 6 patch and 2 control runs · log_lock_waits: silent · tracer stretched the convoy (≈3× slow requests); patch: no measurable cost

Case 3 of 5 · TPC-B at 1,500 tx/s; a backup-like job writes to the same disk 2 s in every 10

Disk bursts look like lock contention

pg_wait_sampling: active sessions, in bursts vs calm

Lock:transactionid



LWLock:WALWrite



IO:WalSync, the flush itself



→ “row-lock contention”

Trace: every WAL flush (IO:WalSync), calm → burst

avg **0.42 → 1.74 ms** **x4**

p99 **0.85 → 8.2 ms** **x10**

calls **2.7× fewer**

→ “the disk got slower”

Fewer but 4× longer flushes: their share barely moves. No bursts: p99 4.2 ms.

pgbench -b tpcb-like, 16 clients, fixed 1,500 tx/s, 120 s · fio randwrite 64k, qd 32 × 4, 2 s of every 10 s · sampler history polled every 1 s · same numbers from pg_wait_tracer (PG18) and the patch (20devel)

Case 4 of 5 · the same primary-key lookup, slowed down two different ways, 60 s each

Slow disk or bloat? The sampler sees twins

	Slow disk	Bloat
Lookup latency, p50	3.9 ms	3.2 ms
pg_wait_sampling	DataFileRead 95%	DataFileRead 89%
pg_wait_tracer: reads	225,780	3,612,309
pg_wait_tracer: avg read	1,964 µs	117 µs
Reads per lookup	1.7	29
Root cause	a noisy neighbour on the disk	an idle transaction blocking cleanup

Measured on PostgreSQL 18, real reads from a real disk: fio flooding the same disk vs. a REPEATABLE READ session idle for 11 minutes while 200k hot rows were updated 30 times · pg_wait_sampling 1.1.11 at 100 Hz

A primary-key lookup that sometimes hit the 1 s statement_timeout

The trace said: storage

~1 ms

a typical DataFileRead

~120 ms

the slow ones, in the tail

9+

slow reads in one lookup: past
1 s

Proof for the storage team, not a hunch. REINDEX helped too: fewer pages per lookup, fewer chances to hit a bad read.

Case 5 of 5 · 4 sessions busy for 60 s under a 0.5-CPU limit (a Kubernetes CPU limit is the same cgroup quota)

Under a CPU limit, the sampler is throttled too

Truth

4.0

active sessions

pg_wait_sampling

0.8–0.9

active sessions, “all CPU”

pg_wait_tracer, exact

4.0

0.4 running + 3.6 waiting for a CPU

Real OLTP at a 1-CPU limit, 72% used, a parallel report every 5 s: p99 4.6 → 317 ms

Profile: WAL 47%, Lock ×3. Nothing says “waiting for a CPU”

Its fix, synchronous_commit = off:
p99 68 ms

Raise the limit to 2 CPUs: **p99 22 ms**

Measured, 4 runs: pg_wait_sampling 1.1.11 saw 20–24% of the time · OLTP: TPC-B 1,400 tx/s, 16 clients, single runs · PostgreSQL 18, 4 vCPU, cgroup v2 cpu.max

A session doing 1 ms of CPU, then a 1 ms sleep, with 32 other sessions on 4 vCPUs

Under load, the wait label outlives the wait

Every wait-event view

75% sleeping

25% CPU

pg_wait_sampling, pg_stat_activity, or a trace of the 4 bytes

The kernel

65% waiting for a CPU

28% sleeping · 7% on a CPU

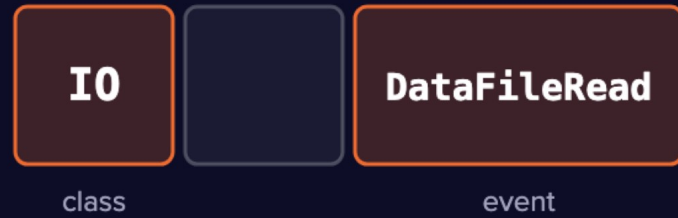
/proc/<pid>/schedstat

A read or lock that has finished keeps its label until the backend gets a CPU. “Slow storage” or “lock contention” can be a CPU queue. Only the scheduler knows.

Measured: PostgreSQL 18, pgbench -S 32 clients, 4 vCPU · run-queue share 0% idle, 27% at 4 clients, 47% at 16, 65% at 32

Every wait is two writes to four bytes

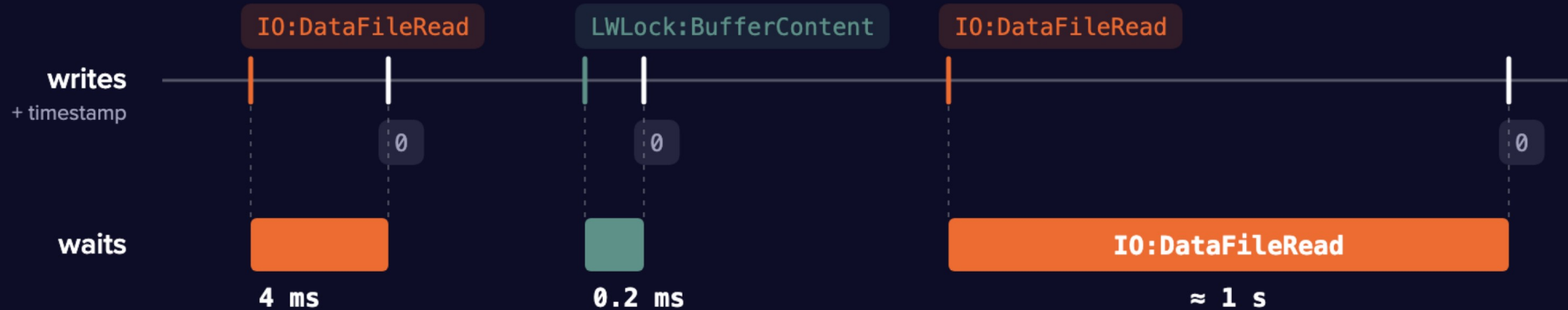
MyProc->wait_event_info



4 bytes in shared memory, one per backend

Every wait starts with a write here,
and ends with a write of zero.

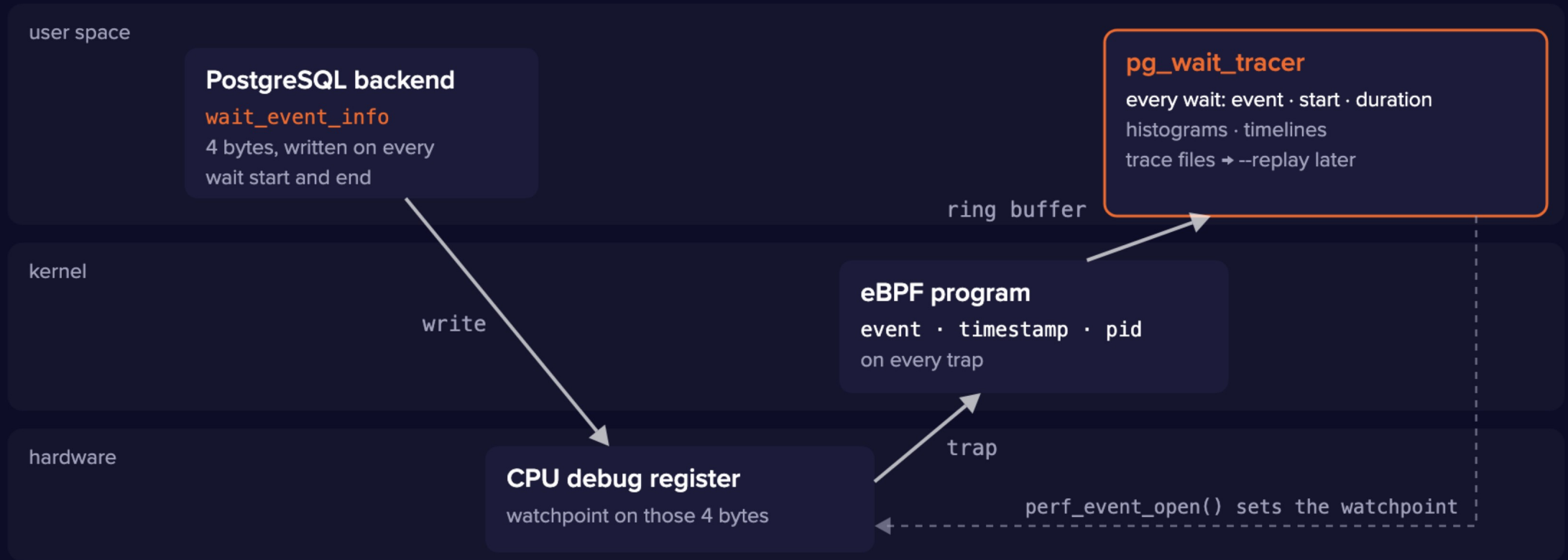
Catch every write, with a timestamp, and you have every wait.



Not a photo. The whole film.

illustration, not to scale

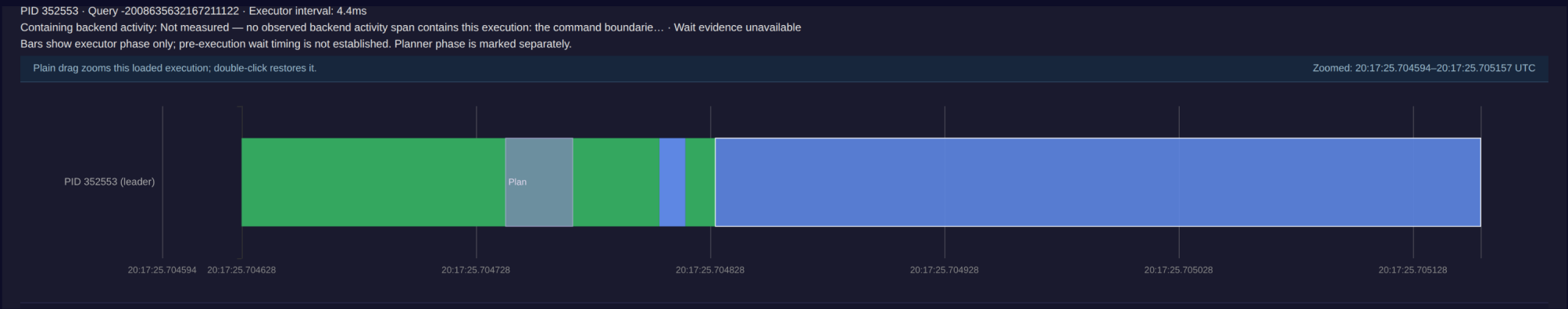
A watchpoint on the four bytes, eBPF on the trap



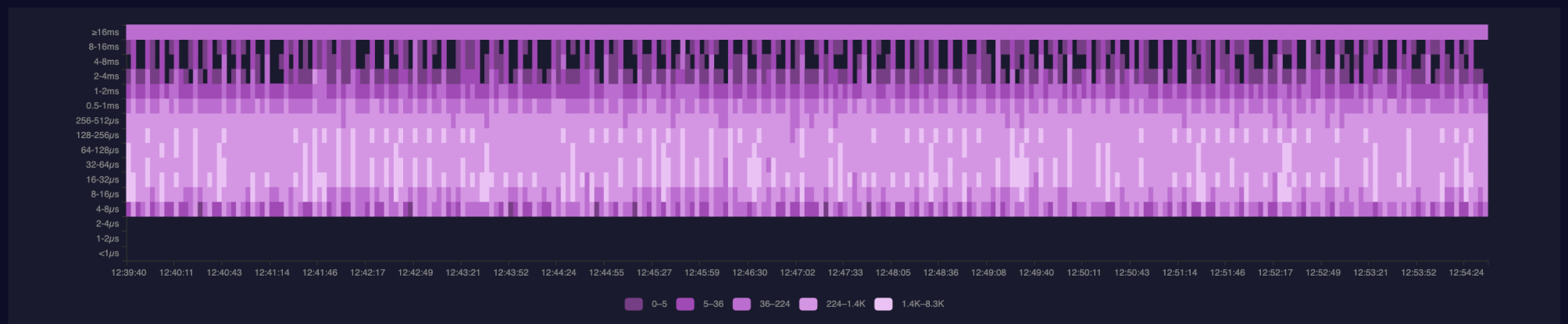
No patch to PostgreSQL. Linux, root or CAP_BPF · CAP_PERFMON · CAP_SYS_PTRACE

Events, not photos

One execution, wait by wait, at microsecond scale



Wait-duration histogram over time, from $<1\ \mu\text{s}$ to $\geq 16\ \text{ms}$



Sample always, trace when it matters

always on · sampled

~0%

on PostgreSQL. Reads wait_event_info from user space at 10 Hz; the daemon uses ~0.6% of one core.

escalated · exact

6–29%

TPS cost while it runs: 6% write-heavy OLTP, 29% read-heavy with fast cache misses.

Escalate by hand or on an anomaly rule, capped at **300 s per hour** by default. The cost is CPU traps, ~5 µs each on cloud VMs, so it grows with the wait rate: in the five cases, from no visible effect to 3× more slow requests.

pgbench, 8 clients, Hetzner cx43 (8 vCPU, 16 GB), PostgreSQL 18 · default mode: --mode tiered · trap cost: 4.7–5.2 µs per watchpoint fire, AMD EPYC KVM guests

Running it

```
# on the DB server
sudo ./pg_wait_tracer --daemon \
-T /var/lib/pgwt/traces
# from your laptop, no root
pgwt root@db-server
# after the incident
pg_wait_tracer --replay \
-T /var/lib/pgwt/traces --from 1h
```

Needs	
PostgreSQL	17, 18 · 13 supported · 14–16 not yet
OS	Linux 5.8+ with BTF, or EL8 · x86_64, aarch64
Privileges	root, or CAP_BPF + CAP_PERFMON + CAP_SYS_PTRACE
Not on	RDS, Cloud SQL, Azure: no host access

Stop guessing. Record every wait.



pg_wait_tracer

github.com/DmitryNFomin/pg_wait_tracer

Try it

pg_wait_tracer on GitHub



Review the patch

commitfest.postgresql.org/patch/6984

Review the patch

Wait timing in core · Commitfest 6984



Connect

linkedin.com/in/dmitry-fomin-a88b7b45

Connect

Dmitry Fomin on LinkedIn

Thank you · Dmitry Fomin