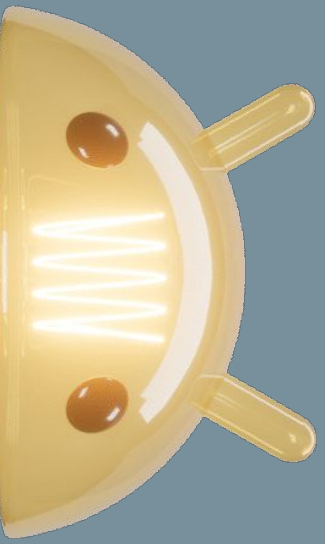


Android

Perfetto recipes

For Linux kernel development

Zimuzo Ezeozue

Software Engineer, Google

- Android OS engineer at Google for 8 years
- Currently building tools to help other engineers diagnose performance issues from the field.
- Perfetto power user and contributor

Table of contents

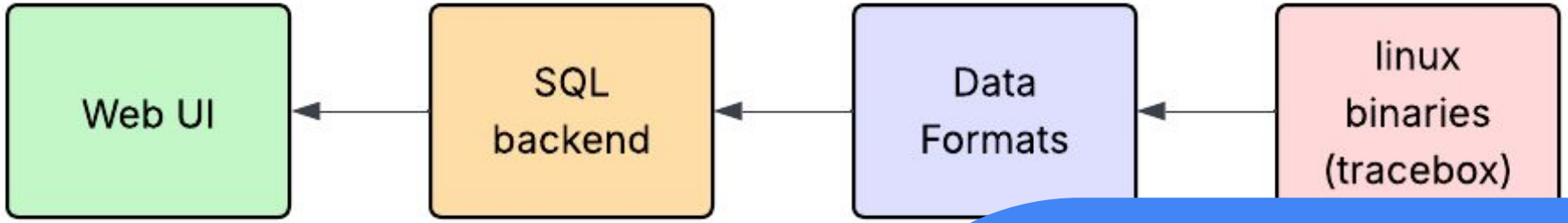
Introduction

Who is this talk for?

Everyone! Really, hopefully everyone learns something about something.

Getting perfetto setup, running a basic config to capture sched and IRQs,

What is Perfetto?

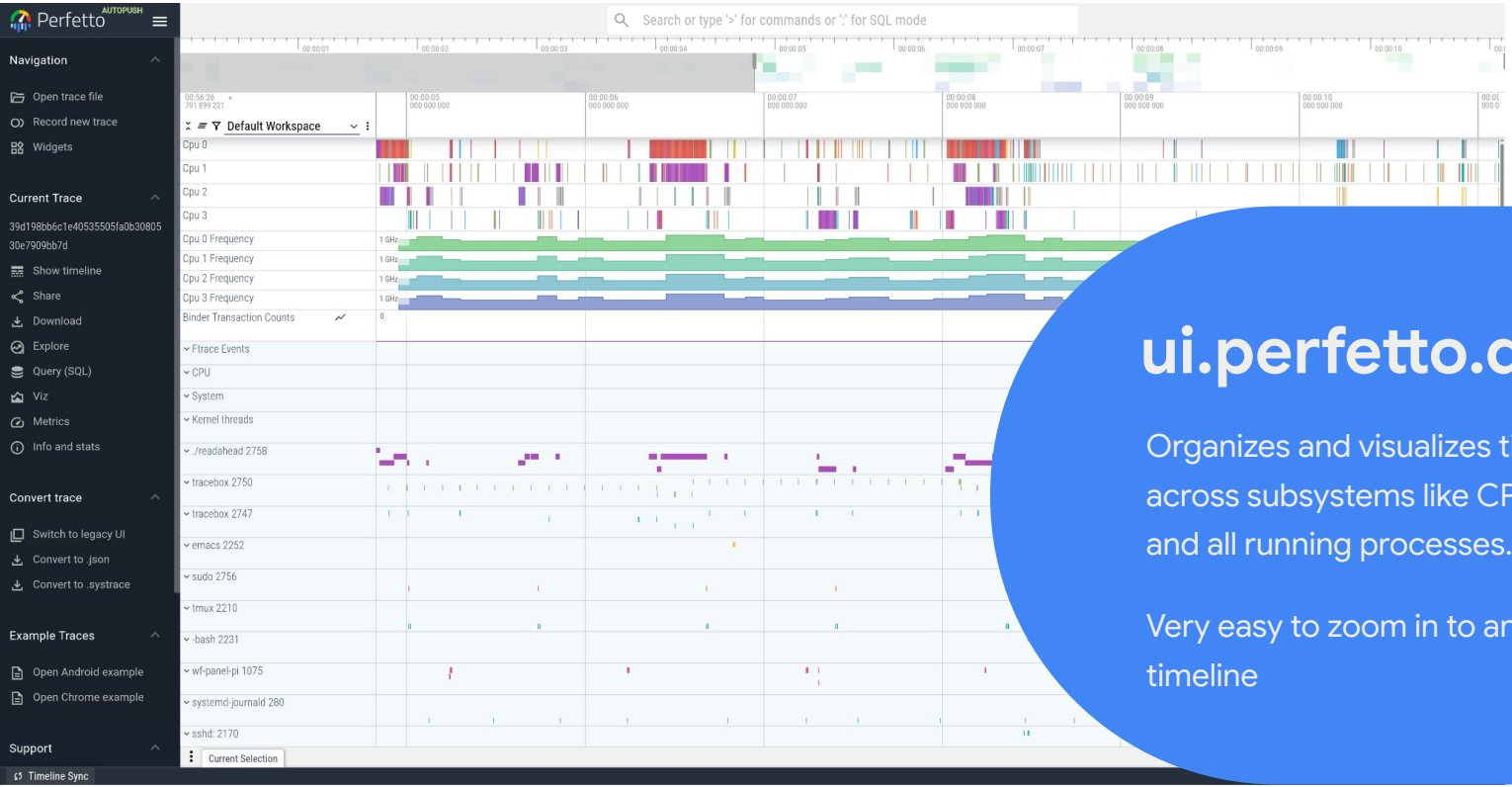


perfetto.dev

Perfetto excels at capturing and visualizing time series data, making it easy to search, aggregate, and derive insights from the information.

Perfetto

Web UI

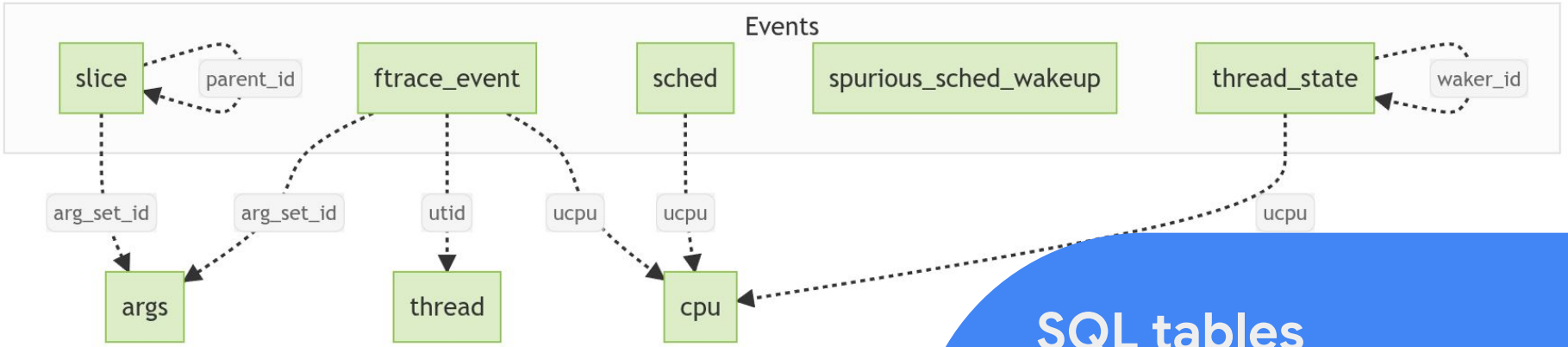


ui.perfetto.dev

Organizes and visualizes time series events across subsystems like CPU, IRQ, Memory and all running processes.

Very easy to zoom in to any region of the timeline

SQL backend



SQL tables

Powers the UI frontend and is directly queryable. In addition to standard SQL functions, it comes with custom functions for interval and graph operations. New functions can also be implemented in C++..

Data formats

- Perfetto native data format
- Android systrace
- Ftrace text output
- JSON
- perf

Supported formats

Two ways to use Perfetto:
1) Record events on device and capture Perfettos rich data format
2) Import data from some other supported format.

These files can all be opened in the UI and there's good support for adding new data formats.

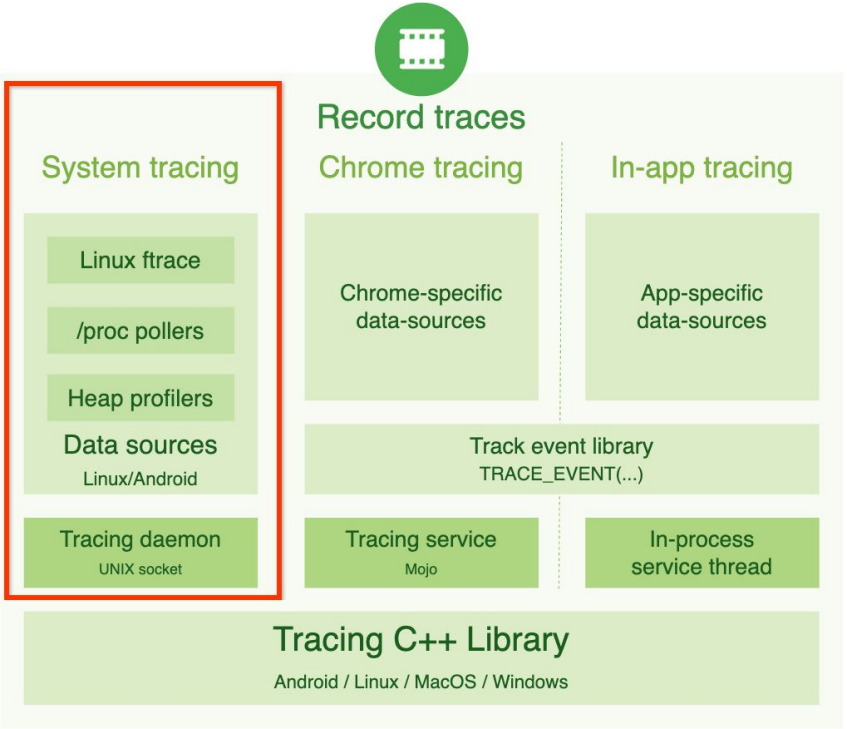
Linux binaries

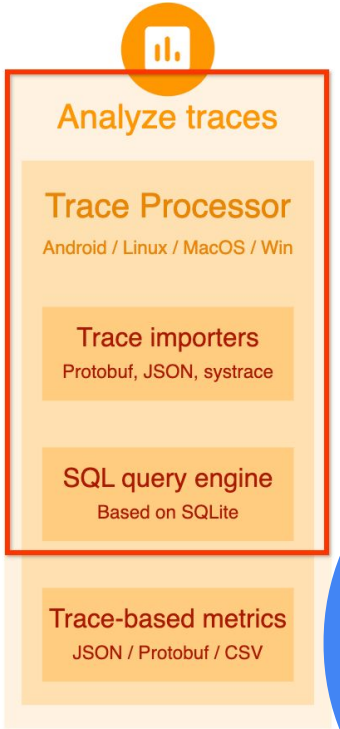
- Collection of processes running on the device that can read events from **tracefs, procfs, sysfs** and many other sources.
- Writes the data into the standardized Perfetto native data format.
- Wrapped behind one binary called **tracebox**
- All the data sources enabled can be configured ahead of time in a simple text file

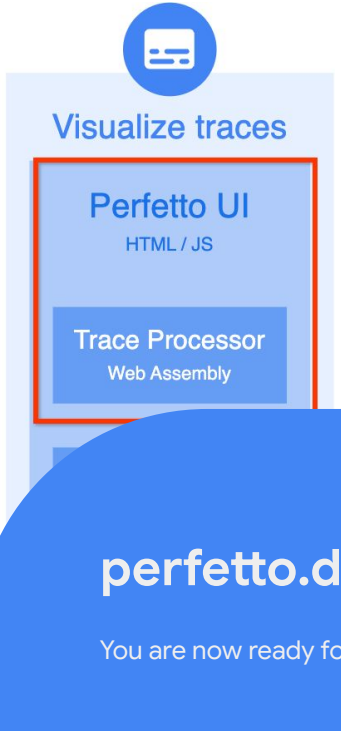
Configs formats

The binaries take care of configuring the data sources: size of buffers to write into, frequency to poll, buffer overrun policy and many more knobs.

Now you are an expert







perfetto.dev/docs

You are now ready for the full detail

Getting started

Download a release

Grab the latest release for your arch from Github:

https://github.com/google/perfetto/releases

Official build instructions

Follow the build instructions from the official perfetto docs:

https://perfetto.dev/docs/contributing/build-instructions

Unofficial instructions

Follow John Stultz' gist for some more detail:

https://gist.github.com/johnstultz-work/0ec4974e0929c4707bfd89c876ae4735

Hello world

Buffers

```
buffers {
  size_kb: 1048576
  fill_policy: RING_BUFFER
}
```

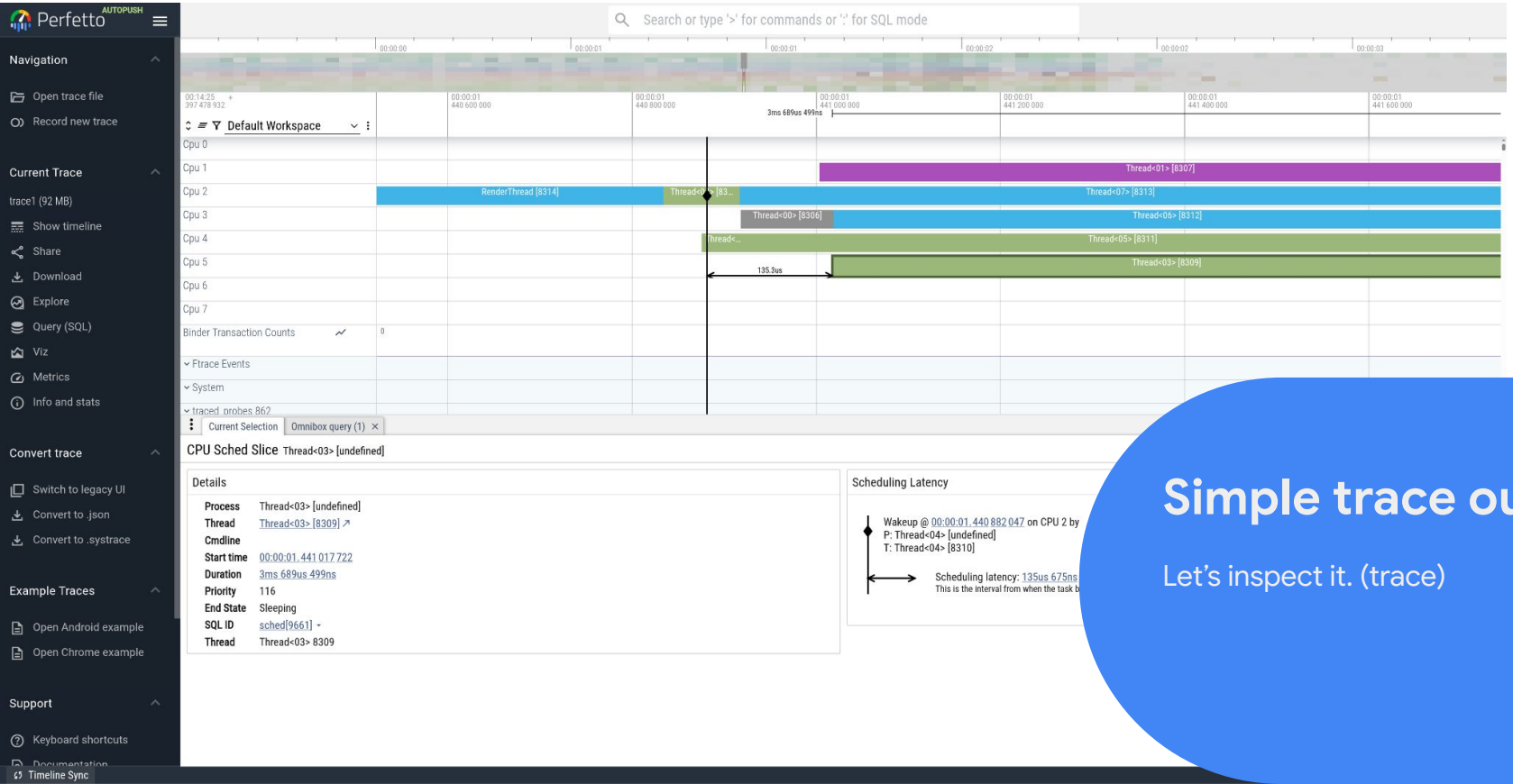
Data sources (producers)

```
data_sources {
  config {
    name: "linux.ftrace"
    target_buffer: 0
    ftrace_config {
      ftrace_events: "sched/sched_switch"
      ftrace_events: "sched/sched_waking"
    }
  }
}
```

Simple config

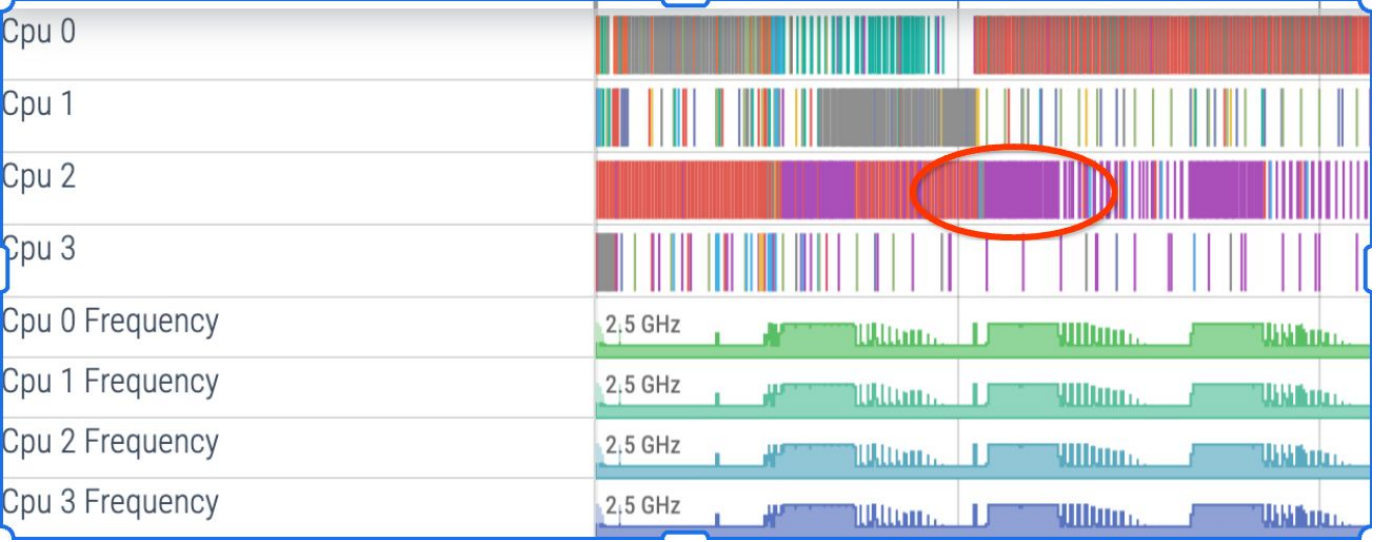
./tracebox --txt -c <config> -o <output file>

Hello world output



Simple trace output

Let's inspect it. (trace)



Frequency scaling

Can see the cpufreq taper off as work frequencies slow. (trace)

Function Graph!

Function graph tracing can be enabled to gain more insight into what's going on. Though this has to be done carefully, as its very easy to overflow the trace buffers perfetto collects. (trace)

Demo: Cyclictest

```
root@jstultz-NUC11TNHi5:/home/jstultz/cyclictest-latency# ./cyclictest -t -p99 -D 15 -q
# /dev/cpu_dma_latency set to 0us
T: 0 ( 2412) P:99 I:1000 C:  15000 Min:      3 Act:    4 Avg:    4 Max:       9
T: 1 ( 2413) P:99 I:1500 C:  10000 Min:      4 Act:    6 Avg:    4 Max:       6
T: 2 ( 2414) P:99 I:2000 C:   7500 Min:      3 Act:    5 Avg:    4 Max:       6
T: 3 ( 2415) P:99 I:2500 C:   6000 Min:      4 Act:    4 Avg:    4 Max:       6
T: 4 ( 2416) P:99 I:3000 C:   5000 Min:      4 Act:    4 Avg:    4 Max:      14
T: 5 ( 2417) P:99 I:3500 C:   4286 Min:      4 Act:    5 Avg:    4 Max:       7
T: 6 ( 2418) P:99 I:4000 C:   3750 Min:      4 Act:    4 Avg:    4 Max:       6
T: 7 ( 2419) P:99 I:4500 C:   3334 Min:      4 Act:    5 Avg:    4 Max:       8
```

Cyclictest

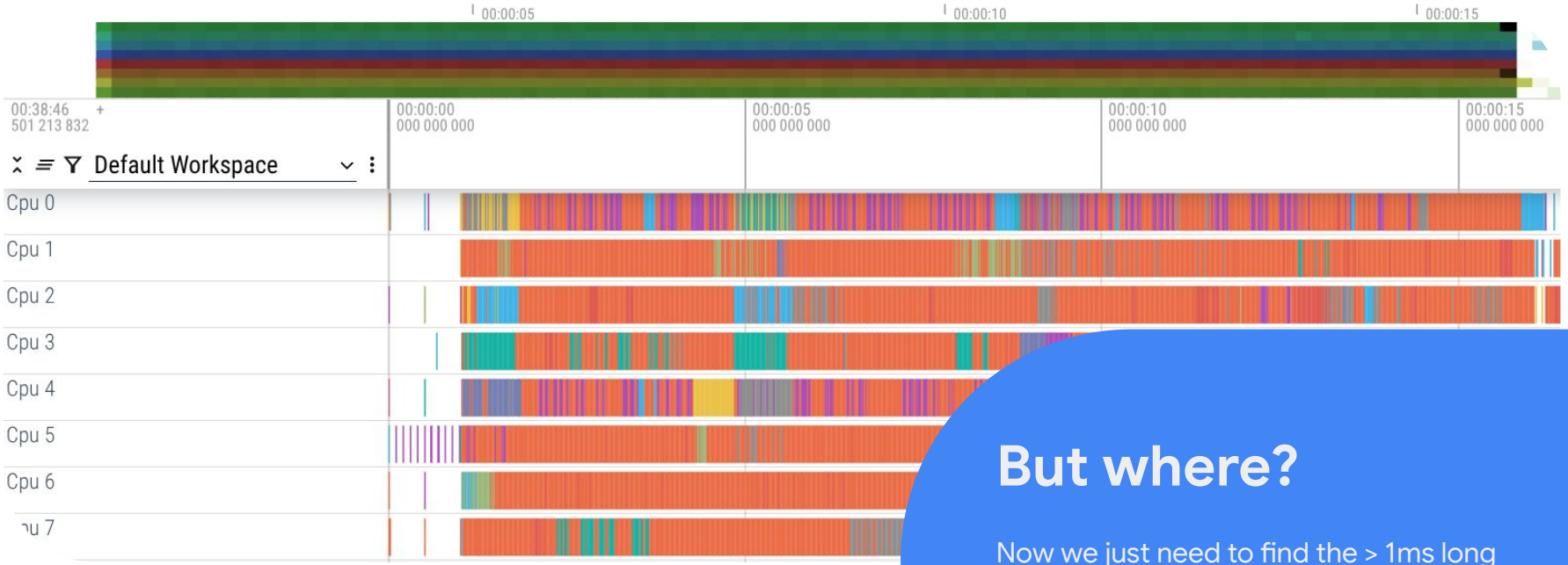
Cyclictest runs RT tasks call nanosleep and measures wakeup latencies, a metric of combined interrupt latency and scheduling latency. Very useful for understanding hardware/OS RT behavior

Cyclictest + Load

More useful to provide background load to exercise kernel and hardware paths. We use this script* to run network and disk load behind cyclictest to find situations that aren't best-case scenarios.

```
/cyclictest-latency# ./test-latency.sh -t 15
# /dev/cpu_dma_latency set to 0us
iperf done
T: 0 ( 2442) P:99 I:1000 C:  15000 Min:      5 Act:    9 Avg:   10 Max:      330
T: 1 ( 2446) P:99 I:1500 C:  10000 Min:      5 Act:   10 Avg:   11 Max:      756
T: 2 ( 2450) P:99 I:2000 C:   7500 Min:      6 Act:    9 Avg:   13 Max:     1053
T: 3 ( 2453) P:99 I:2500 C:   6000 Min:      6 Act:    8 Avg:   16 Max:      292
T: 4 ( 2455) P:99 I:3000 C:   5000 Min:      7 Act:   14 Avg:   13 Max:     1103
T: 5 ( 2457) P:99 I:3500 C:   4286 Min:      7 Act:   11 Avg:   15 Max:      288
T: 6 ( 2459) P:99 I:4000 C:   3750 Min:      5 Act:   15 Avg:   13 Max:      935
T: 7 ( 2460) P:99 I:4500 C:   3334 Min:      6 Act:   10 Avg:   13 Max:      540
```


*https://github.com/johnstultz-work/cyclictest-latency



But where?

Now we just need to find the > 1ms long run in 15 seconds of trace over 8 cpus. (trace)

query and press Cmd/Ctrl + Enter

```
1  SELECT * FROM thread_state LEFT JOIN thread using(utid) WHERE
2  state='R' AND name='cyclictest' AND dur > 1000000
```

Query result (28 rows) - 39.6ms SELECT * FROM thread_state LEFT JOIN thread using(utid) WHERE … Showing rows 1 to 28 of 28 Prev Next Copy

id	ts	dur	cpu	utid	state	io_wait	blocked_function	waker_utid	waker_id	irq_context	ucpu	id_1	tid	name	start_ts
681	2327546006981	1003634	NULL	406	R	NULL	NULL	433	539	1	NULL	406	2129	cyclictest	2327531622194
1117	2327553877561	1367101	NULL	414	R	NULL	NULL	420	1105	1	NULL	414	2137	cyclictest	2327535510128
6282	2327597366903	4005602	NULL	406	R	NULL	NULL	433	6239	1	NULL	406	2129	cyclictest	2327531622194
9137	2327632491275	2750859	NULL	406	R	NULL	NULL	433	9030	1	NULL	406	2129	cyclictest	2327531622194
40972	2328019150109	1494469	NULL	406	R	NULL	NULL	418	40967	1	NULL	406	2129	cyclictest	2327531622194
559491	2329765216811	1193817	NULL	406	R	NULL	NULL	418	559206	1	NULL	406	2129	cyclictest	2327531622194
711652	2330230714315	1168482	NULL	406	R	NULL	NULL	418	711623	1	NULL	406	2129	cyclictest	2327531622194
1018382	2331141005398	2009255	NULL	406	R	NULL	NULL	439	1018271	1	NULL	406	2129	cyclictest	2327531622194
1083917	2331396228342	1671876	NULL	406	R	NULL	NULL	429	1083887	1	NULL	406	2129	cyclictest	2327531622194
1087475	2331468381108	1459493	NULL	406	R	NULL	NULL	429	1087468	1	NULL	406	2129	cyclictest	2327531622194
1091013	2331520178659	1574455	NULL	406	R	NULL	NULL	429	1090998	1	NULL	406	2129	cyclictest	2327531622194
1113496	2331864313159	1457557	NULL	406	R	NULL	NULL	429	1113486	1	NULL	406	2129	cyclictest	232753162219
20393	2331976574172	1062111	NULL	406	R	NULL	NULL	429	1120306	1	NULL	406	2129	cyclictest	2327531622
67	2334479270251	1108863	NULL	406	R	NULL	NULL	426	1945018	1	NULL	406	2129	cyclictest	23275316

Further filtering...

Unfortunately, Cyclictest has one housekeeping SCHED_NORMAL thread that we expect to be delayed, so filter out that one thread by tid.

And we are left with just one instance.

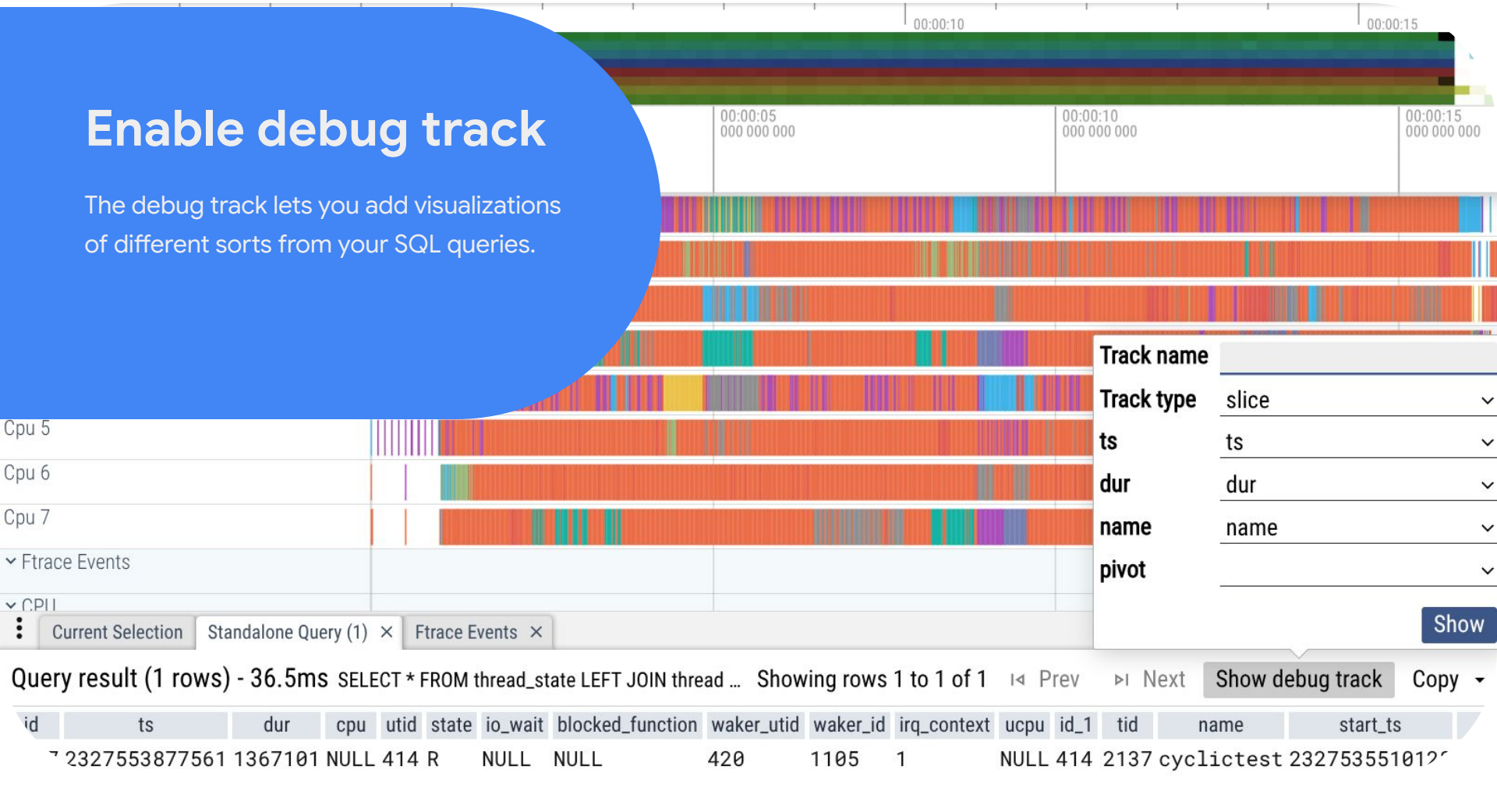
```
1  SELECT * FROM thread_state LEFT JOIN thread using(utid) WHERE
2  state='R' AND name='cyclictest' AND dur > 1000000 and tid != 2129
```

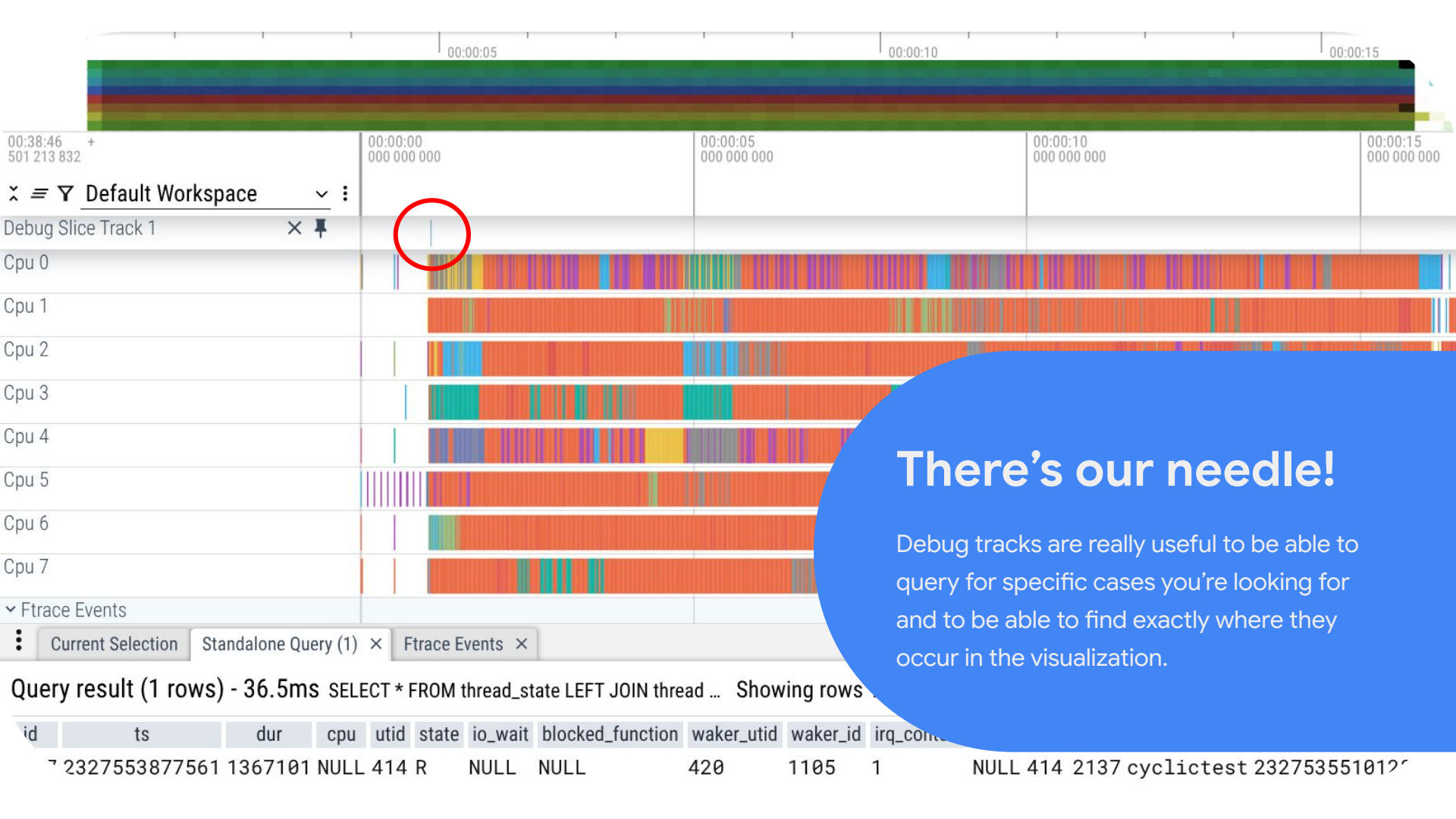
Query result (1 rows) - 36.5ms SELECT * FROM thread_state LEFT JOIN thread using(utid) WHERE state... Showing rows 1 to 1 of 1 Prev Next Copy

id	ts	dur	cpu	utid	state	io_wait	blocked_function	waker_utid	waker_id	irq_context	ucpu	id_1	tid	name	start_ts	
1117	2327553877561	1367101	NULL	414	R	NULL	NULL	420	1105	1	NULL	414	2137	cyclictest	2327535510128	234

Enable debug track

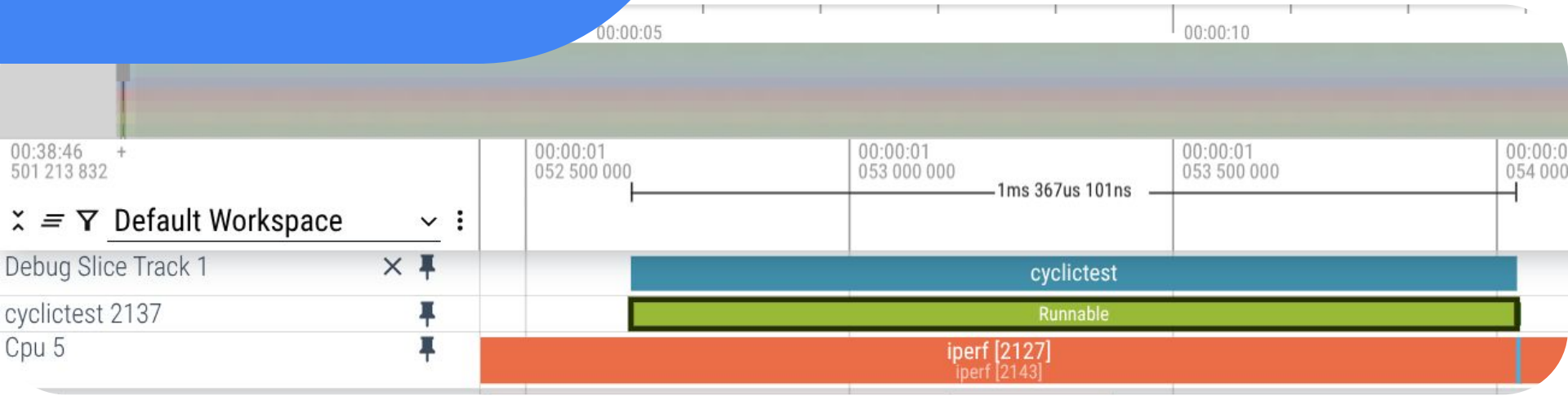
The debug track lets you add visualizations of different sorts from your SQL queries.

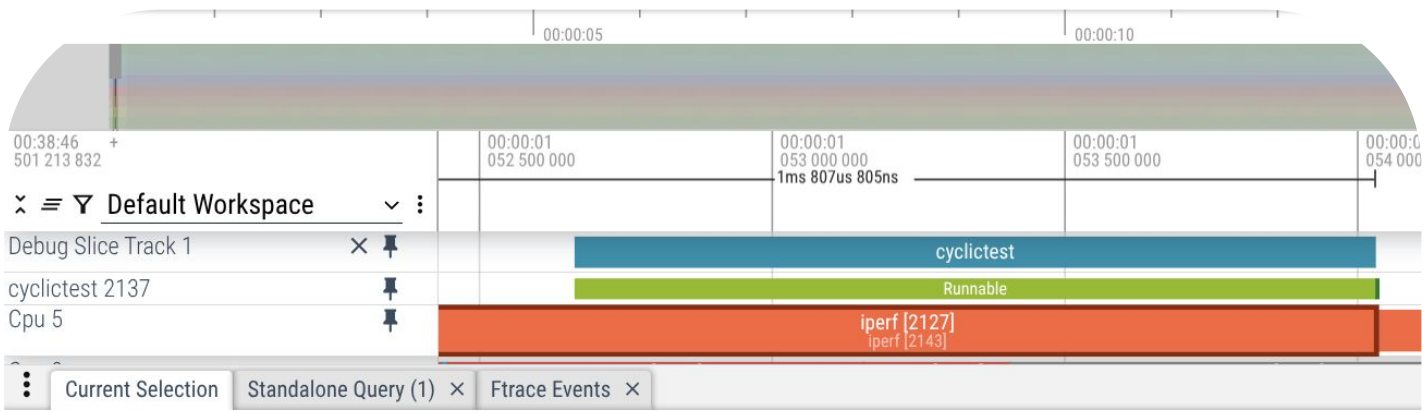



00:00:05
00:00:10
00:00:15
00:38:46
501 213 832
00:00:00
000 000 000
00:00:05
000 000 000
00:00:10
000 000 000
00:00:15
000 000 000
Default Workspace
Debug Slice Track 1
Cpu 0
Cpu 1
Cpu 2
Cpu 3
Cpu 4
Cpu 5
Cpu 6
Cpu 7
Ftrace Events
Current Selection
Standalone Query (1)
Ftrace Events
Query result (1 rows) - 36.5ms
SELECT * FROM thread_state LEFT JOIN thread ...
Showing rows
id
ts
dur
cpu
utid
state
io_wait
blocked_function
waker_utid
waker_id
irq_cont
2327553877561 1367101 NULL 414 R NULL NULL 420 1105 1 NULL 414 2137 cyclictest 232753551012
There’s our needle!
Debug tracks are really useful to be able to query for specific cases you’re looking for and to be able to find exactly where they occur in the visualization.

Zooming in

We can see the RT99 prio cyclictest has been runnable but not scheduled for over 1ms





CPU Sched Slice iperf [2127]

Details

Process	iperf [2127]
Thread	iperf [2143] ↗
Cmdline	iperf -c localhost -t 15 -i 1 -P 8
Start time	00:00:01.052 223 025
Duration	1ms 807us 805ns
Priority	120
End State	Runnable (Preempted)
SQL ID	sched[513]
Thread	iperf 2143
Process	iperf 2127
`er ID	0

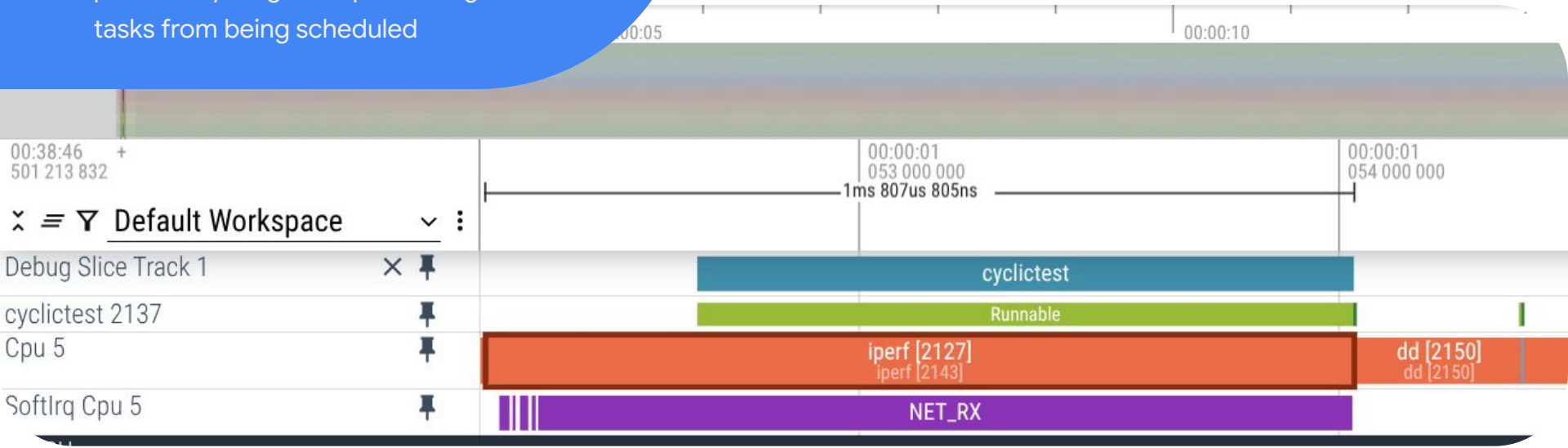
Strange!

Iperf is a SCHED_NORMAL (prio 120) process, so it should be immediately preempted when a RT99 task is woken! It should not block a RT99 task!

Check softirq track

Looking at the softirq track, we find our culprit.

A network softirq is running for a particularly long time, preventing userland tasks from being scheduled



Demo: Instrumenting userspace

```
uint64_t run_read_test(const char *filename, size_t block_size, int num_blocks,
                       int use_readahead, const char *test_name) {
    trace_begin(test_name, (size_t)block_size * num_blocks);

    int fd = open(filename, O_RDONLY);
    if (!use_readahead)
        posix_fadvise(fd, 0, 0, POSIX_FADV_RANDOM);
    read(num_blocks, block_size);

    trace_end();
    return throughput;
}
```

Simple benchmark

Contrived example but is simple enough to illustrate the power of perfetto. What are these trace_begn() and trace_end()?

```
void trace_begin(const char* name, size_t size) {
    if (trace_marker_fd >= 0) {
        char buffer[256];
        sprintf(buffer, "B|%d|%s|size=%zu", getpid(), name, size);
        write(trace_marker_fd, buffer, strlen(buffer));
    }
}

void trace_end() {
  if (trace_marker_fd >= 0) {
    char buffer[63];
    sprintf(buffer, "E|%d", getpid());
    write(trace_marker_fd, buffer, strlen(buffer));
  }
}
```

Instrumenting userspace

Perfetto natively understands these Begin and End markers in the trace buffer.

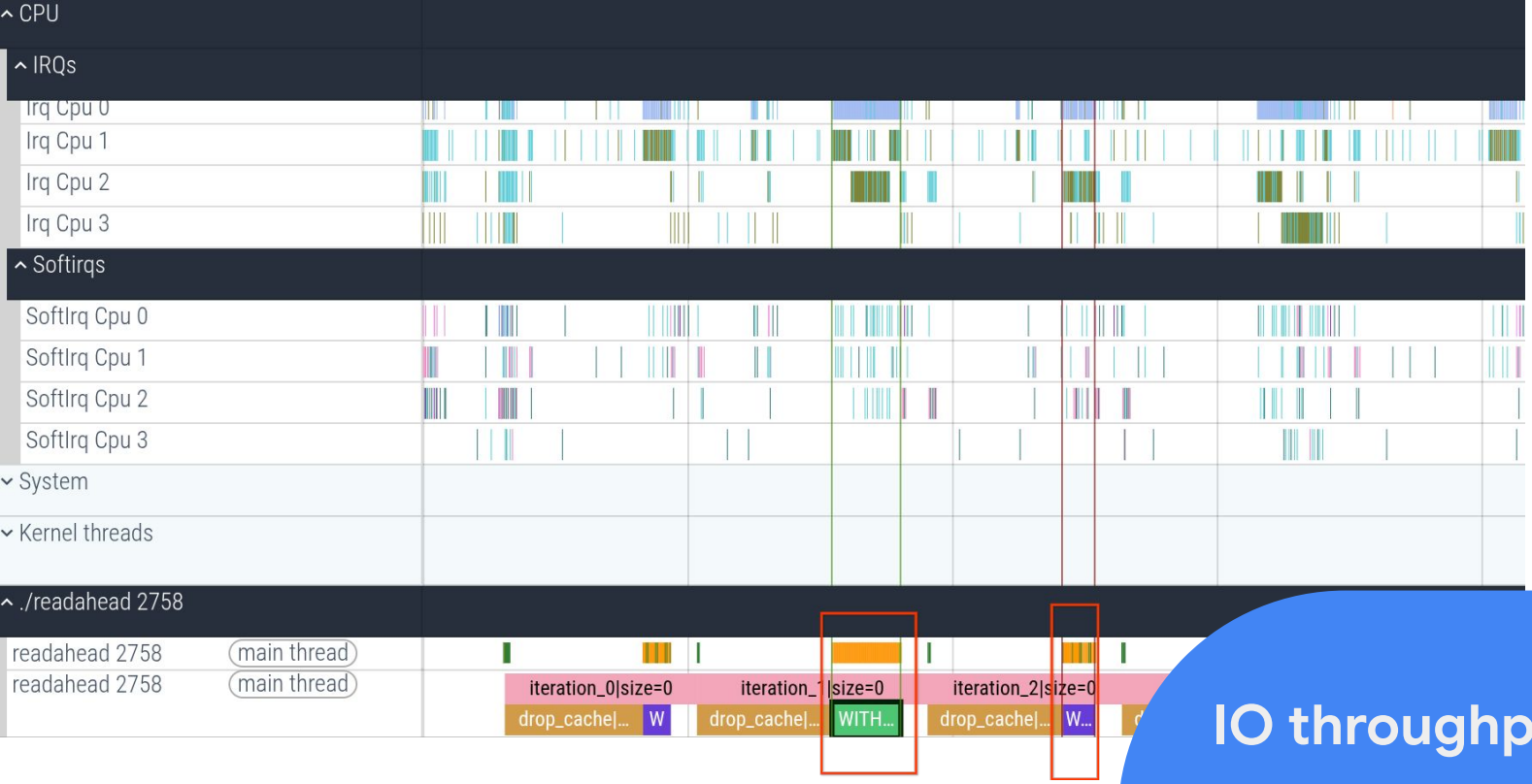
The perfetto sdk is a more robust for serious instrumenting but requires more than one slide

```
efiat-stable:~/dev/perfetto-configs $ sudo ./readahead
Using existing file: test_file (3 MB)
WITH_READAHEAD: 105369 us, Read: 4096000 bytes, Throughput: 37 MB/s
WITHOUT_READAHEAD: 241884 us, Read: 4096000 bytes, Throughput: 16 MB/s
WITH_READAHEAD: 102538 us, Read: 4096000 bytes, Throughput: 38 MB/s
WITHOUT_READAHEAD: 235268 us, Read: 4096000 bytes, Throughput: 16 MB/s
WITH_READAHEAD: 99856 us, Read: 4096000 bytes, Throughput: 39 MB/s
WITHOUT_READAHEAD: 248633 us, Read: 4096000 bytes, Throughput: 15 MB/s
WITH_READAHEAD: 102925 us, Read: 4096000 bytes, Throughput: 37 MB/s
WITHOUT_READAHEAD: 300973 us, Read: 4096000 bytes, Throughput: 12 MB/s
WITH_READAHEAD: 105473 us, Read: 4096000 bytes, Throughput: 37 MB/s
WITHOUT_READAHEAD: 305538 us, Read: 4096000 bytes, Throughput: 12 MB/s

===== SUMMARY =====
WITH_READAHEAD:    Avg: 37 MB/s  Max: 39 MB/s
WITHOUT_READAHEAD: Avg: 14 MB/s  Max: 16 MB/s
MISLEADING OVERALL: Avg: 25 MB/s
ni@fiat-stable:~/dev/perfetto-configs $
```

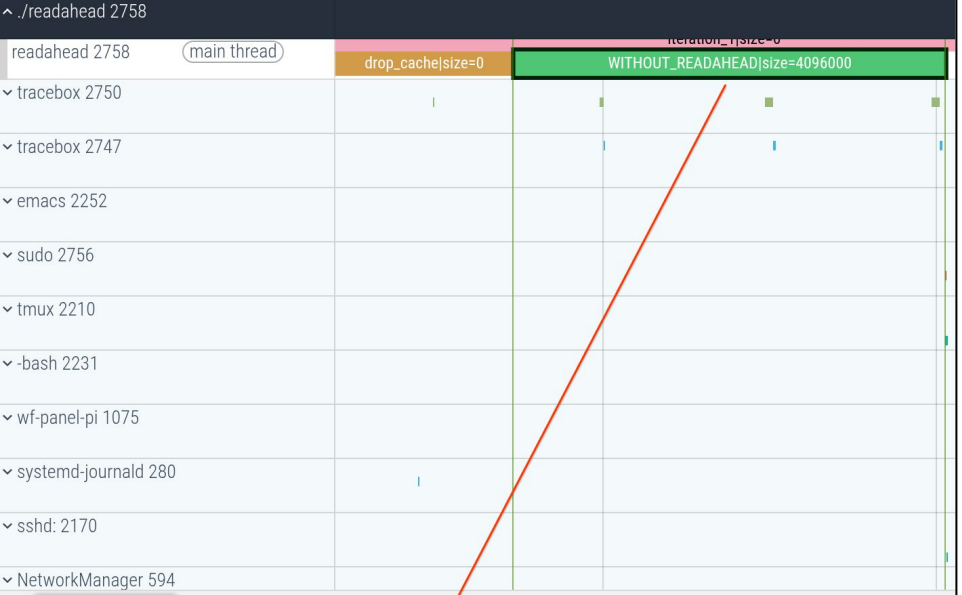
IO benchmark

Contrived workload to illustrate the power of instrumenting userspace and visualizing userspace and kernel events on the same timeline



IO throughput

Can see the bimodal behavior along the same timeline as IRQS and other system events. (trace)



Current Selection

Slice WITHOUT_READAHEAD|size=4096000

Details

Name	WITHOUT_READAHEAD|size=4096000
Category	N
Start time	0
Absolute Time	2025-05-06T00:49:17.575712084
Duration	259ms 974us 591ns
Uninterruptible Sleep	190ms 976us 428ns (73.46%)
Runnable (Preempted)	23ms 131us 939ns (8.90%)
Running	31ms 378us 897ns (12.07%)
Runnable	14ms 487us 327ns (5.57%)
Thread	readahead [2758]
Process	./readahead [2758]
User ID	0

Slices with the same name

UI Drill down

Easy to find similar slices in the UI

Table Slices

× name = 'WITHOUT_READAHEAD|size=4096000'

id	ts	dur ↓	category	name
24781 ↗	00:00:08.024 187 250	333ms 870us 955ns	*NULL*	WITHOUT_READAHEAD|size=4096000
18754 ↗	00:00:06.360 825 779	319ms 144us 928ns	*NULL*	WITHOUT_READAHEAD|size=4096000
13181 ↗	00:00:04.740 545 343	270ms 50us 948ns	*NULL*	WITHOUT_READAHEAD|size=4096000
7378 ↗	00:00:03.150 341 683	264ms 861us 484ns	*NULL*	WITHOUT_READAHEAD|size=4096000
1287 ↗	00:00:01.546 422 044	259ms 974us 591ns	*NULL*	WITHOUT_READAHEAD|size=4096000

Table Slices

× name = 'WITH_READAHEAD|size=4096000'

id	ts	dur ↓	category
6609 ↗	00:00:02.417 137 214	122ms 442us 592ns	*NULL*
18291 ↗	00:00:05.628 835 382	107ms 636us 441ns	*NULL*
668 ↗	00:00:00.831 122 229	104ms 865us 15ns	*NULL*
12738 ↗	00:00:04.025 898 084	104ms 74us 782ns	*NULL*
24022 ↗	00:00:07.301 598 760	103ms 22us 569ns	*NULL*

Comparison

Can clearly see the readahead is consistently faster

Enter query and press Cmd/Ctrl + Enter

```
SELECT name, thread_slice.dur / 1e6 AS total_time_ms, SUM(thread_state.dur) / 1e6 AS running_time_ms
FROM thread_slice
JOIN thread_state
  USING (utid)
WHERE
  name LIKE '%READAHEAD%'
  AND thread_state.ts BETWEEN thread_slice.ts AND thread_slice.ts + thread_slice.dur
  AND thread_state.state = 'Running'
GROUP BY thread_state.state, thread_slice.dur
ORDER BY name, total_time_ms DESC
```

Query result (10 rows) - 54.5ms SELECT name, thread_slice.dur/1e6 AS total_time_ms, SUM(thread_state.dur)/1e6 AS running_time_ms FROM thread_slice JOIN thread_state USING(utid) WHERE name LIKE '%READAHEAD... Showing rows 1 to 10 of 10 Prev

name	total_time_ms	running_time_ms
WITHOUT_READAHEAD|size=4096000	333.870955	65.310365
WITHOUT_READAHEAD|size=4096000	319.144928	62.148402
WITHOUT_READAHEAD|size=4096000	270.050948	38.723209
WITHOUT_READAHEAD|size=4096000	264.861484	31.271411
WITHOUT_READAHEAD|size=4096000	259.974591	31.250268
WITH_READAHEAD|size=4096000	122.442592	6.944177
WITH_READAHEAD|size=4096000	107.636441	17.978727
WITH_READAHEAD|size=4096000	104.865015	
WITH_READAHEAD|size=4096000	104.074	
WITH_READAHEAD|size=4096000	103	

Time spent Running

Advanced query to showcase the power of the SQL engine

Thank you

Questions?

Appendix

Links

Traces

- Hello world trace: https://ui.perfetto.dev/#!/?s=d7328d7f6d087fe8838def3bffc722a975d597b1
- Freq scaling trace: https://ui.perfetto.dev/#!/?s=d451f58815578682d704b009e3fc713ab8d59400
- Function graph trace: https://ui.perfetto.dev/#!/?s=c7ac4b85e7d90ed3e8174330ad99ab27403f68e1
- Cyclic test trace: https://ui.perfetto.dev/#!/?s=61bb2e239e2c06b10b5e50f34c9f08de34b5dfef
- Userspace trace: https://ui.perfetto.dev/#!/?s=5d45da63a75108be73752e2af4ebb06f5e5b1a72

Scripts

- Cyclic test script: https://github.com/johnstultz-work/cyclictest-latency
- Readahead script: https://github.com/zezeozue/readahead-demo/blob/main/readahead.c