Pi 4 Linux Embedded Workload Tracing

Jerry Qi

December 2025

Abstract

I present a linux kernel module for lightweight, online monitoring of real time scheduling behaviour. The module instruments scheduler and interrupt tracepoints to track worst case latency, response time, control loop timing, and IRQ handling time for selected tasks in an embedded scenario. To minimize overhead, statistics are updated using lock free atomics on the hot path, with trace collection performed only on rare worst case events. We explicitly include less costly synchronization primitives to ensure max/min/counter and trace correctness, and intentionally abstain from using heavy synchronization. We evaluate the module's overhead using `ktime` and `cyclictest` and show it doesn't significantly impact real time scheduling latency under typical workloads. The tool enables rapid diagnosis of real time (RT) health without requiring full trace capture or offline analysis.

1 Motivation

The vision of this tool is to allow people to run this on an existing Linux system that cares about real time execution, and see **quickly** if the system is in a healthy real time state.

Given a particular running workload on a Pi 4, and that we have the custom kernel (tracepoint patch) running with the `jerry_rt_module.ko` binary inside, we can get worst case numbers and scheduler traces for latency, response time, control loop response time, and IRQ response time relatively easily.

The core point of this application is specific to verifying scheduling / RT qualities in a quicker manner than browsing the entire trace-cmd or kernelshark logs (which is a lot and not constrained to relevant scheduler events). If there is a particular oddity, we might quickly pinpoint which task is the root of the issue.

This can be helpful for applications that are time critical. For example rockets that need a motor turned in a particular moment. Verifying that latency is low allows one to know the task is being run in a short time. Verifying that response time is low allows one to know that the task is being finished in a short time.

2 Design and why

2.1 Basic Mechanics

2.1.1 What is collected

As mentioned briefly above, the application allows the tracking of max/min metrics for a task:

1. Latency of schedule-in: Wakeup $\rightarrow$ Running

2. Response time to first voluntary sleep
3. Response time for a cycle of a control loop
4. IRQ Handler run time

1 allows us to know about the degree of competition from other tasks. 2 and 3 are measurements to gauge how long a particular event/periodic task takes to "finish a response" to some signal. 4 measures the handler function run time for a single IRQ line pull (Given the task id is an IRQ thread).

Along with max/min metrics, we allow one to set a upper bound for each particular metric (e.g. $U_{latency}$). A violation of such bound by S_1 allows a micro event trace to be generated for $U_{latency} + S_1$. We output the worst violation trace. Given $S_2 > S_1$ and S_2 and S_1 being the only violations, the trace output will correspond to $U_{latency} + S_2$.

This results in output like this:

```
[177173.922241] jerry_rt_module: PID=9459, Name=control_loop_bl, minLat=3074, maxLat=34593,
[177173.922241] minResponseTimeVoluntarySleepAllTypes=15204,
maxResponseTimeVoluntarySleepAllTypes=1103092,
[177173.922241] minResponseTimeVoluntarySleepReliefBased=6970926,
maxResponseTimeVoluntarySleepReliefBased=7109982
[177173.922272]   LatencyBound=100, Violations=112608

[177173.972583] WORST LATENCY TRACE:
[177173.978756] [     0 µs] Event: sched_wakeup, pid: 9459, wake_cpu: 1
[177173.987997] [     3 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 9460
(priority: 49), next_pid: 0 (priority: 120), on_cpu: 2,
[177174.004042] [    26 µs] Event: sched_wakeup, pid: 174, wake_cpu: 1
[177174.013231] [    34 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
(priority: 120), next_pid: 9459 (priority: 9), on_cpu: 1,
```

Note that currently, units are not uniform across the output vs trace. The output of raw statistics is in nanoseconds. The trace time range uses µs as labelled.

We also see that the `WORST LATENCY TRACE` corresponds to the `maxLat`.

2.1.2 Basic Mechanisms

The module works on a per task ID basis, the module storing a `task_latency_entry` per id. Inside each pid struct, we store max/min statistics along with utility variables, along with violation and trace buffers.

We register a few tracepoints: `sched_wakeup`, `sched_switch`, `sys_enter`, `sched_process_exit`, `irq_threaded_handler_entry`, `irq_threaded_handler_exit` (`irq_threaded_handler_entry/exit` are custom kernel patch introduced tracepoints).

Using these tracepoints we track max/mins for a pid accordingly:

1. Latency for task: Its `sched_wakeup` - `sched_switchin`
2. Response time voluntary: Its `sched_wakeup` - voluntary `sched_switch` out

3. Response time relief: Its `sched_wakeup` - timer based `sched_switch` out
4. IRQ based response time: A pid's IRQ exit - A pid's IRQ entry

Additionally, when there is a violation X to our set SLO bound and the violation is larger than the previous S, $S < X$, then we have a trace collection onto a ring buffer.

At the end, on pid removal (potentially triggered by task exit, manual removal, or module removal), all relevant statistics for pid is outputted.

2.2 Overview of Model and Logic

2.2.1 Latency Model

This is trivial; simply the `sched_wakeup` to the `sched_switch` in for a pid.

2.2.2 Response time voluntary Model

This is a task's `sched_wakeup` to the voluntary `sched_switch` out for a pid. When a task gets un-scheduled from a CPU, it is either because of a voluntary switch out or a involuntary context switch (preemption). To track voluntary-ness, we check whether the task after switching out, enters one of `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_PARKED`, `TASK_IDLE`, `TASK_DEAD` states.

There is a reason we use the `unsigned int prev_state` rather than the `bool preempt` provided in the kernel's `sched_switch` probe call argument.

The tracepoint′s preempt flag is best thought of as: "was prev switched out while still runnable (preempted) vs did it stop being runnable (voluntary). It doesn't necessarily mean preempt=1 is always the case for higher priority switch ins.

For example, a higher priority RT task woke up, this triggers a `schedule()` call, which entails `sched_switch` of the lower priority task; this tracepoint call would have `preempt=1`. On the other hand, say a task blocks, and calls `schedule()`, potentially a higher priority task woke up at the same time and can be scheduled in, the `sched_switch` here would `preempt=0`.

2.2.3 Response time Relief Model

This metric is based on a typical embedded device control loop. Here is one such possible FSM as illustrated in Figure 1. However, there could be different variations based on this (e.g. Rockets have control loops that don't have blocking operations, therefore no voluntary sleeps, and retrieve telemetry information based on IRQs handlers). As long as the control loop is periodic and using a timer sleep model per iteration (i.e. handling an "event" per iteration), the metric is applicable.

To track the time from `sched_wakeup` to timer based `sched_switch`, we explicitly ignore preemption based and other voluntary based switch outs and only care about timer based switch outs. The nontrivial part here is the timer based `sched_switch` out. To separate this with other types of voluntary sleeps, we explicitly try to detect a `sleep()` or `nanosleep()` invocation via `sys_enter` (both are syscalls). A post sleep voluntary `sched_switch` indicates such a timer based switch out.

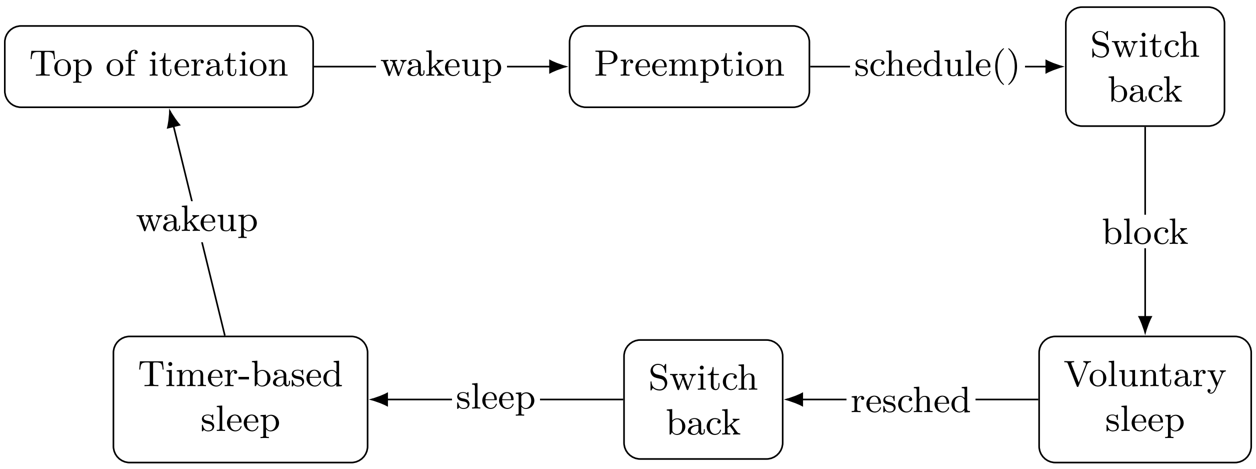

Figure 1: Control-loop FSM

2.2.4 Race identification

Each pid has a dedicated `task_latency_entry`, so state associated with different pids is naturally partitioned and does not race. Therefore potential races arise only within the state of a single pid.

At a high level, a task may only execute on one CPU at a time, so scheduler events such as `sched_switch` for a given task are emitted from a single core at a single time. However, this property alone is insufficient to eliminate races in the module. Tracepoint handlers execute synchronously on the CPU that triggers the tracepoint, but handlers corresponding to different events for the same pid may overlap in time across CPUs. For example, a wakeup event on one CPU and a subsequent scheduling event for the same pid on another CPU may be processed concurrently. For a latency example, let a wakeup record a wakeup time on core 1, and let a `sched_switch` in be on core 2 happen very fast, before core 1's wakeup handler finishes processing. On a single core, tracepoint handlers execute serially, but across CPUs they may execute concurrently, so races arise only from multicore concurrency.

As a result, concurrent updates to per pid statistics (e.g., max/min values, violation counters, and trace metadata) are possible and must be explicitly synchronized. This is talked about more in Section 7.

3 Usage

As mentioned above, the core features are max/min capturing, and violation SLO setting and tracing.

3.1 Load the module

```
insmod jerry_rt_module.ko
```

After this line, the module is loaded. However, no pids are tracked yet.

Note: replace `jerry_rt_module.ko` with your kernel binary's path.

3.2 Track pids

To start tracking particular pids in a real time environment, we can either track it to include tracing or not.

```
echo "123" > /sys/kernel/jerry_rt_module/add_pid
```

Figure 2: Tracking pid 123 without trace

```
echo "123 trace" > /sys/kernel/jerry_rt_module/add_pid
```

Figure 3: Tracking pid 123 with trace

3.3 Tracking Pids with Violations and Trace

After starting the tracking for a particular pid, we can set an upper bound (SLO) for a particular metric. Let such an upper bound be X, we'd establish a count for the number of times the metric exceeds X from the moment the upper bound is set until tracking ends.

```
echo "123 5000" > /sys/kernel/jerry_rt_module/set_slo_latency_bound
```

Figure 4: Setting a latency upper bound for pid 123

We use format "[pid] [slo bound in ns]".

To set the bounds for the other statistics: `set_slo_response_bound`, `set_slo_response_relief_bound`, `set_slo_irq_handling_bound` can be used.

Note that if one enabled tracing from 3.2, one also needs to set a SLO bound to actually get a trace.

3.4 Ending Tracking

There are several methods to ending tracking for a pid.

Removing a single pid

```
echo "123" > /sys/kernel/jerry_rt_module/del_pid
```

Figure 5: Removing tracking for a single pid 123, broadcasting all its statistics accumulated before this removal command

Detaching the module

```
rmmod jerry_rt_module.ko
```

Figure 6: Untracks all pids, broadcasts all statistics of all pids and removes module

For the special case where a pid terminates by itself, we have registered a `sched_process_exit` tracepoint to broadcast statistics on its exit.

3.5 A cumulative example

```
# echo "22543 trace" > /sys/kernel/jerry_rt_module/add_pid
# echo "22543 2" > /sys/kernel/jerry_rt_module/set_slo_latency_bound
# echo "22543 2" > /sys/kernel/jerry_rt_module/set_slo_response_bound
$ echo "22543 2" > /sys/kernel/jerry_rt_module/set_slo_response_relief_bound

... Let some time pass for program to run ....

# echo "22543" > /sys/kernel/jerry_rt_module/del_pid
```

Figure 7: Adds pid with tracing; sets a latency bound; sets a response bound; set a response relief bound; Let workload run; We untrack the single pid and get its statistics

The statistics that we get for pid can be seen below

```
[233048.284704] jerry_rt_module: PID=22543, Name=background_bloc, minLat=3519, maxLat=25482,
[233048.284704] minResponseTimeVoluntarySleepAllTypes=1017797,
↪  maxResponseTimeVoluntarySleepAllTypes=1059093,
[233048.284704] minResponseTimeVoluntarySleepReliefBased=9223372036854775807,
↪  maxResponseTimeVoluntarySleepReliefBased=0
[233048.284734]   LatencyBound=2, Violations=19434
[233048.330618] MAX TRACE LEN: 3
[233048.336594] WORST LATENCY TRACE:
[233048.342939] [     0 us] Event: sched_wakeup, pid: 22543, wake_cpu: 1
[233048.352448] [     3 us] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 22544
↪  (priority: 49), next_pid: 0 (priority: 120), on_cpu: 2,
[233048.368766] [    25 us] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
↪  (priority: 120), next_pid: 22543 (priority: 69), on_cpu: 1,
[233048.385067]   ResponseBound=2, Violations=18267
[233048.392707] MAX TRACE LEN: 3
[233048.398638] WORST RESPONSE TIME TRACE:
[233048.405503] [     0 us] Event: sched_wakeup, pid: 22543, wake_cpu: 1
[233048.415018] [ =    7 us] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
↪  (priority: 120), next_pid: 22543 (priority: 69), on_cpu: 1,
[233048.431315] [  1059 us] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 22543
↪  (priority: 69), next_pid: 0 (priority: 120), on_cpu: 1,
[233048.447570]   ResponseReliefBound=2, Violations=0
[233048.455299] MAX TRACE LEN: 0
[233048.461148] WORST RESPONSE RELIEF TIME TRACE:
```

Figure 8: Statistics format from the pid 22543

This information can help understand what is going on with the workload. Note that the response relief trace for this workload doesn't exist because it is event driven rather than periodic.

3.6 Inprecision of userspace control plane

There are several points in the module where the exact boundary at which recording begins or ends is intentionally not precisely defined. These arise from the interaction between user-triggered control operations (e.g., adding or removing a pid, setting SLO bounds) and asynchronously executing tracepoint handlers.

Specifically, the following questions do not have a strict answer:

1. When does statistic recording begin after issuing the `start_recording_pid` command?
2. When do SLO violation checks begin to reflect a newly configured bound?

These ambiguities do not stem from a lack of cache coherence or global visibility—all writes are eventually visible on all CPUs—but rather from the absence of explicit synchronization between userspace control plane updates and the timing of tracepoint execution on other CPUs.

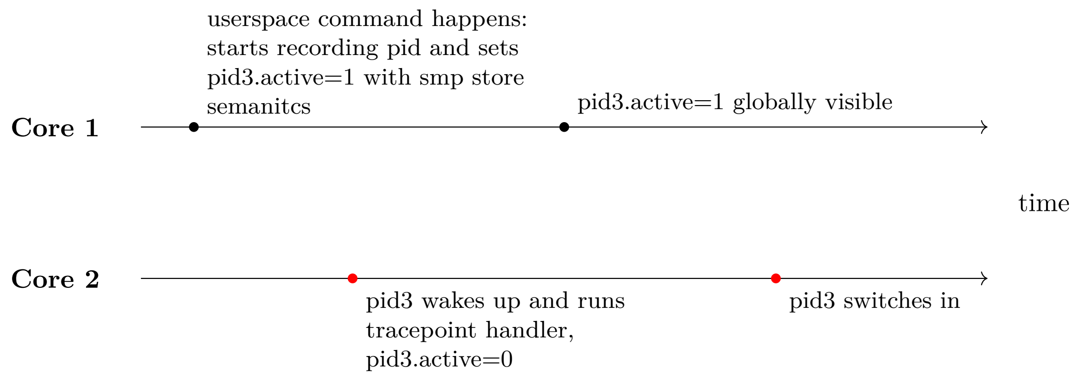
userspace command happens: starts recording pid and sets pid3.active=1 with smp store semanitcs
pid3.active=1 globally visible
Core 1
time
Core 2
pid3 wakes up and runs tracepoint handler, pid3.active=0
pid3 switches in

Figure 9: Missed initial latency cycle, despite cycle being later in time than pid track command

For pid tracking addition, the `active` flag is published using a release store. This guarantees ordering once the store is observed, but doesn't guarantee when other cores observe it. As a result, a tracepoint corresponding to a latency cycle may execute before observing `active == true`, even if the userspace command to begin tracking occurred earlier. In this case, the initial cycle is not recorded.

Pid removal uses an atomic `xchg` on the `active` flag, which provides global visibility once it completes. Nevertheless, tracepoint events that occur before the atomic transition executes may still observe `active == true` and update statistics. Consequently, a small number of events occurring shortly after the userspace removal command may still be included.

SLO bound updates follow a similar pattern to pid addition. Bounds are written without synchronizing against concurrent tracepoint execution, so violations may be checked against either the old or new bound depending on timing. This behavior is acceptable, as SLO configuration is a low frequency control operation and exact boundary precision is not required.

This design is chosen because the module aims to reduce as much overhead from the original workload as possible, therefore having less synchronization. This causes exact timing of starts and ends of max/min recording and SLO violation ranges to be a bit imprecise, though, not to the degree of seconds. I would like to implement xchg for Pid addition in a later version, as using xchg tightens the window between the userspace and when other CPUs *must* observe the change (after `xchg` finishes, other CPUs must observe the new value).

4 Impact on original workload

There are a few elements in the module that contributes to the overhead in a normal run.

1. **Global cost per tracepoint hit**
   - Every `sched_switch`, `sched_wakeup`, `sys_enter`, and IRQ entry/exit, there is an additional handler call, a few loads/stores, `ktime_get` call, and the ring buffer push.
2. **Cost per tracked tracepoint hit**
   - If a tracked tracepoint was hit then there is extra cost for bookkeeping. (i.e. xchg, CAS loop, reads/writes)
3. **Rare slow path: trace collection**
   - When a new maximum is observed, the module acquires a per-metric lock and walks the trace ring buffer via `collect_slo_trace_common`. This path is heavier but occurs infrequently, only when a new worst-case metric is detected.
4. **Start/Stop tracking**
   - Sysfs add/del (Rare).

Tracepoint handlers execute synchronously on the same CPU and in the same execution context as the traced event. Therefore, each tracepoint hit directly adds latency to the kernel path that triggered it. We refer to this always incurred cost as the *base cost* C_b. Say a switch in previously took X, with no pid that is tracked, it now takes $X + C_b$.

When a PID is actively tracked, additional per event bookkeeping is performed (e.g. max/min updates), which we refer to as the *tracked cost* C_t. Say a switch in/out event occured for a pid that we care about, then cost $= C_b + C_t + X$. The base cost dominates overall overhead, as it applies to all tracepoint hits regardless of how many PIDs are being tracked.

4.1 Evaluation Approach

Since the most significant costs arise from the global tracepoint overhead and the tracepoint overhead once pids are tracked, we mainly focus on those for this evaluation.

We evaluate both the per-event execution cost and the impact on real time scheduling behavior.

4.1.1 Per Event Global Overhead

First we quantify the per event overhead on the system imposed by the extra tracepoint probe. We vary when the module is loaded, and when there are pids tracked.

$$\text{avg\_cost} = \frac{\sum \text{handler execution time over all CPUs and PIDs}}{\text{total number of tracepoint hits}}$$

avg_cost gives the global amortized cost per event. For example, if only one PID is tracked, its extra tracking work is "diluted" by all the untracked events.

Experimental Method

Experiments were run on a `PREEMPT` Linux kernel on a Pi 4 with 3 fixed periodic workloads running. We compared module loaded with no tracked PIDs, and module loaded with one or more tracked PIDs. Each run lasted 30 seconds. The module records total tracepoint invocations and total handler execution time, from which average per event overhead was computed. The maximum time spent in a handler (probe) is also recorded.

Configuration	Average ns / Probe	Max ns in probe
0 PIDs tracked	463	1889
1 PID tracked	638	4913
2 PIDs tracked	824	6006
3 PIDs tracked	997	11080

Table 1: Tracepoint handler overhead under different tracking configurations (30s run).

Tracking additional PIDs increases the average tracepoint handler latency (Average ns/probe) in a predictable manner. Even with three tracked PIDs, the average overhead remains below 1 µs per probe.

The overhead scales approximately linearly with the number of tracked PIDs, indicating that the module's cost is dominated by per-PID bookkeeping rather than global contention or serialization effects.

The observed worst case probe latencies (Max ns/probe) increase faster than linearly with the number of tracked PIDs. These outliers are likely attributable to occasional slow paths such as synchronization and trace buffer updates, which are exercised only when a new maximum is detected. Despite this, the worst case probe runtime remains below 12 µs with 3 tracked PIDs, which is within an acceptable range for many embedded and soft real time workloads operating at microsecond granularity.

4.1.2 Evaluation of impact on RT-ness

Second, we use cyclictest to evaluate whether this added per event overhead impacts real time scheduling behaviour.

We run cyclictest with a fixed duration (120 seconds) alongside three existing background periodic loops of a range of priorities: normal, `SCHED_FIFO` 70, `SCHED_FIFO` 79. As in the first experiment, all background tasks and cyclictest are pinned to one core to maximized scheduling contention. We vary if the module is loaded and cyclictest is tracked and observe whether the latencies gathered increases as a result. This helps us understand the effect of the module on RT latency for a single pid.

Configuration	Min	Avg	Max (µs)
cyclictest no module	4	6	24
cyclictest with module	5	7	28.3
cyclictest with module tracked	5.3	7.7	29.3

Table 2: Cyclictest latency results under different module configurations

For the third config, to register tracking immediately after cyclictest spawns, we use

```
cyclictest -D 120 -p 80 -i 1000 -m -t1 -a 2 > ./experiment_2_out/cyclictest.out 2>&1 & PID=$!;
RT_TID=$((PID+1)); echo "$RT_TID trace" > /sys/kernel/jerry_rt_module/add_pid
```

in the assumption that the cyclictest thread has tid = cyclictest_pid + 1 (not guaranteed but working for my machine).

As seen by Table 2, the module did indeed cause latencies to increase across min, avg and max. However, the µs increase is not significant enough to affect real time guarantees. For a 1 ms cyclictest period, the observed increase remains well below 5% of the scheduling interval and does not affect worst case latency.

Additional from tracking the cyclictest thread, we can track other pids to see whether the additional overhead affects latency numbers in cyclictest, the highest priority task. We keep tracking the cyclictest thread, and add additional pids that are tracked.

We essentially keep the previous setup, however, now we start tracking the three background periodic loops using the module.

Configuration	Min	Avg	Max (µs)
additional 1 pid tracked	5	7	26
additional 2 pids tracked	5	8	25
additional 3 pids tracked	5	8	24

Table 3: Cyclictest latency results with additional pids tracked

Table 3 shows cyclictest latency when additional periodic loop tasks are tracked by the module. While the average latency increases slightly (1 µs) when tracking two or more additional PIDs, the worst case latency does not increase and remains comparable across configurations. This indicates that additional per PID tracking overhead does not meaningfully impact the scheduling latency of the highest priority real time task. In particular, no degradation in worst case latency is observed, which is the primary metric for real time correctness.

5 Space Evaluation

This kernel module does take a bit of space to store data for pids when running. The dominant contributor is the PID indexed table `pidtab` containing `struct task_latency_entry` objects.

`struct task_stat_slo` has four metric blocks (latency, response, response_relief_bound, irq_handling_bound).

Each metric block takes 32 bytes (s64 (8) + `atomic_t` (4) + padding (4) + int (4) + int (4) + pointer (8)), so, each struct is 128 bytes.

Accounting for all remaining fields in `struct task_latency_entry`, including timestamps, state flags, and four `raw_spinlock_t` instances (sized according to the kernel configuration), the total size of each entry is approximately 296 bytes.

Globally, we have `PIDTAB_SIZE=65536` task latency entries allocated, then we have $65536 \times 296 = 19,398,656$ bytes $\approx$ 18.5 MiB

The module maintains a PID indexed table of `task_latency_entry` structs. The design intentionally trades memory footprint for quick lookup. By indexing directly using a PID, we avoid more complex structures such as dynamic allocation and hash lookups, which would introduce additional latency into tracepoint paths.

Although the resulting memory footprint is 18.5 MiB for a full PID range on a Raspberry Pi 4 running a 64 bit kernel, this cost is incurred only once at initialization. Importantly, this memory overhead does not add per event cost or interfere with scheduling behavior during execution. This overhead is acceptable on modern embedded systems that are more lax with memory and aligns with the goal of minimizing runtime interference.

6 Case Studies

Remark that the goal with this tool is not to specifically identify the bug, but to show that the trace reduces the failure search space to a small, actionable set of causes. Here I show the tool's capability through a variety of scenarios.

6.1 Scenario 1: Misconfigured Priorities in a RT System

We simulate a real time control loop that executes every 2ms and performs a fixed amount of non-blocking computation. This task is latency critical: missing deadlines would degrade system reliability significantly.

At the same time, a background task performs non critical (real time) computation (e.g. sensor aggregation or logging). Under correct configuration, this task should never interfere with the control loop's scheduling.

To evaluate whether the tool can diagnose scheduler level interference, we intentionally introduce a priority misconfiguration.

6.1.1 Test Configuration

Both tasks are pinned to the same CPU to eliminate load balancing effects.

```
taskset -c 2 chrt -f 80 ./control_loop_bin
taskset -c 2 chrt -f 90 ./background_bin
```

The background task is deliberately assigned a higher `SCHED_FIFO` priority than the control loop.

6.1.2 Observed results

```
PID=20596 (control_loop_bin)
minLat=3203 ns, maxLat=1,649,908 ns
```

```
LatencyBound=3000 ns, Violations=10195

[85730.686891] WORST LATENCY TRACE:
[85730.692691] [     0 µs] Event: sched_wakeup, pid: 20596, wake_cpu: 2
[85730.701590] [   796 µs] Event: sched_wakeup, pid: 18, wake_cpu: 3
[85730.710185] [   801 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
(priority: 120), next_pid: 18 (priority: 120), on_cpu: 3,
[85730.725691] [   815 µs] Event: sched_switch, preemption: 0, voluntary: 1026, prev_pid: 18
(priority: 120), next_pid: 0 (priority: 120), on_cpu: 3,
[85730.741476] [  1649 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 20598
(priority: 9), next_pid: 20596 (priority: 19), on_cpu: 2,
```

(Note: These are excerpts from the overall output from the module)

The control loop (pid 20596) wakes at t = 0µs, but doesn't execute until 1649µs, exceeding the 3 µs latency bound by 3 orders of magnitude.

6.1.3 Trace Reasoning

The final `sched_switch` event shows that:

1. The control loop (PID 20596) was scheduled only after PID 20598 voluntarily yielded
2. PID 20598 has a higher real time priority than the control loop (higher ranking → lower number)
3. No intervening `sched_switch` involving PID 20598 appears in the trace, indicating it ran continuously during our entire violation window (scheduled in before)

From this trace we can rule out:

- IRQ interference - no IRQ threads causing our control loop to switch out
- CPU migration - control loop remains on CPU2
- Lock contention - our control loop is runnable by definition of `sched_wakeup`

The trace evidence directly identifies the root cause: a higher priority real time task is monopolizing CPU 2, potentially due to sustained computation or insufficient yielding that prevents the control loop from being scheduled. This allows us to quickly narrow down to a priority configuration error without requiring searching through a ftrace dump, manually querying for scheduler stats, or trial and error tuning.

The same latency based reasoning applies to other forms of CPU contention (e.g. lots of competing RT tasks) without requiring a separate case study.

6.2 Scenario 2: Accidental Blocking in RT System

We simulate a real time control loop similar to Case study 1. The loop executes every 5 ms and performs approximately 1 ms of computation per iteration. As before, the task is latency and deadline critical, meaning that prolonged execution or blocking would impact system correctness.

Unlike the previous case, no external competing tasks are intentionally introduced. Instead, we **introduce a second thread within the same process** that guards a shared resource using a semaphore. The control loop attempts to acquire this semaphore during each iteration, simulating access to a critical internal resource such as a shared queue or buffer.

This setup models a common failure mode in real time systems: **accidental blocking inside a well prioritized control loop**. The goal is to evaluate whether the tool can diagnose issues internal to a single program rather than interference from other tasks.

6.2.1 Task Configuration

Like we said above, we simply introduce a main process running the control loop, which first spawns a "disrupting thread" first before entering the control loop section.

```
taskset -c 1 chrt -f 80 ./control_loop_block_bin
```

6.2.2 Observed Results from trace

```
[177173.922241] jerry_rt_module: PID=9459, Name=control_loop_bl, minLat=3074, maxLat=34593,
[177173.922241] minResponseTimeVoluntarySleepAllTypes=15204,
maxResponseTimeVoluntarySleepAllTypes=1103092,
[177173.922241] minResponseTimeVoluntarySleepReliefBased=6970926,
maxResponseTimeVoluntarySleepReliefBased=7109982

LatencyBound=100, Violations=112608
ResponseBound=1000, Violations=118524
ResponseReliefBound=10000, Violations=61755
```

Our control loop is pid 9459.

(Note: Bounds do not represent expectations and are simply random numbers chosen for the experiment)

From observation, latency does not appear to be bad, having a worst case wake to run time of µs. This number is relatively healthy for a real time control loop task. This indicates that the task likely wakes and runs promptly and there is little scheduler level interference from other tasks. Therefore the issue is not in task priority configuration or CPU contention.

6.2.3 Response Time (any voluntary sleep) analysis

From a wake to a first voluntary sleep, there is a WC of about 1ms. The trace looks like this:

```
[177174.049664] [     0 µs] Event: sched_wakeup, pid: 9459, wake_cpu: 1
[177174.058971] [     4 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
(priority: 120), next_pid: 9459 (priority: 9), on_cpu: 1,
[177174.075009] [  1102 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 9459
(priority: 9), next_pid: 9451 (priority: 120), on_cpu: 1,
```

This indicates the control loop wakes, runs immediately, executes roughly 1 ms of computation, and then voluntarily yields the CPU. This behaviour is expected given the simulated workload and does not yet indicate a correctness issue.

6.2.4 Response Relief time analysis

The most severe behaviour appears in response relief metric, which captures the time from wakeup until the task voluntarily sleeps. The worst case trace is:

```
[177174.112839] [     0 µs] Event: sched_wakeup, pid: 9459, wake_cpu: 1
[177174.122135] [     3 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
(priority: 120), next_pid: 9459 (priority: 9), on_cpu: 1,
[177174.138212] [  1012 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 9459
(priority: 9), next_pid: 0 (priority: 120), on_cpu: 1,
[177174.154268] [  7072 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 9567
(priority: 120), next_pid: 9459 (priority: 9), on_cpu: 1,
[177174.170570] [  7098 µs] Event: sys_sleep, pid: 9459
[177174.178641] [  7109 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 9459
(priority: 9), next_pid: 9567 (priority: 120), on_cpu: 1,
```

The trace shows that after we do about 1ms of execution in the control loop (pid 9459) we voluntarily switch to pid 0, the idle task on `PREEMPT` Linux. The loop remains blocked for 6ms before resuming on the same CPU. This indicates that the task itself initiates a blocking operation rather than getting preempted under influence of other tasks.

From the trace evidence above we can conclude:

- Not scheduler contention - the task runs immediately after waking up and there is no preempting activity in the trace during control loop execution
- Not IRQ interference - no preempting softirq thread activity appears in the trace.
  - While hard IRQ execution itself is not directly captured, prolonged IRQ interference would manifest via softirq threads, which are absent here.
- Not CPU affinity issues - execution resumes on the same CPU on which the task was woken; no core switching scheduling overhead

The only remaining explanation is voluntary blocking inside the control loop itself, consistent with the semaphore wait introduced in this experiment. Correlating the response relief interval with the first voluntary sleep event, we can pinpoint that the blocking happens 1ms into each control loop iteration, matching the duration of the simulated busy workload.

This case demonstrates that low wakeup latency alone is insufficient to guarantee real time responsiveness, internal blocking behaviour must also be explicitly accounted for.

6.3 Scenario 3: Blocking Event-Driven background Task

We now simulate an event driven task, which differs fundamentally from the periodic control loop model assumed in previous case studies. An event driven task would get woken by some sort of signal, do some work (potentially blocking), and then wait for another signal to do the same thing again.

Since the task does not follow a periodic structure and does not explicitly signal iteration completion via a sleep, the concept of a "relief" phase is undefined for this workload. As a result, the response relief metric, designed for measuring wake to relief cycles in periodic loops, is not applicable here.

We place a simulated computation in the middle to emulate the computations of a sample background task, such as tabulating statistics from sensors to log onto an SD card. **We intentionally place a faulty blocking operation which delays event handle time as a semaphore wait**.

6.3.1 Task configuration

A blocking thread is made inside of the task to hold the semaphore of the faulty block. Another event thread is made to arbitrarily simulate pings to our task to wake up and do computation. Both of these threads are set at `SCHED_FIFO` and priority 50, and pinned onto different CPUs, which minimizes interference from other unrelated system activity.

The host task is set with a `SCHED_FIFO` with priority 30.

```
taskset -c 1 chrt -f 30 ./background_block_bin
```

6.3.2 Observed results from trace

```
[233048.284704] jerry_rt_module: PID=22543, Name=background_bloc, minLat=3519, maxLat=25482,
[233048.284704] minResponseTimeVoluntarySleepAllTypes=1017797,
maxResponseTimeVoluntarySleepAllTypes=1059093,
[233048.284704] minResponseTimeVoluntarySleepReliefBased=9223372036854775807,
maxResponseTimeVoluntarySleepReliefBased=0
```

Latency looks like there is no issue with a WC of 25µs response time (Prev case studies).

```
[233048.336594] WORST LATENCY TRACE:
[233048.342939] [     0 µs] Event: sched_wakeup, pid: 22543, wake_cpu: 1
[233048.352448] [     3 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 22544
(priority: 49), next_pid: 0 (priority: 120), on_cpu: 2,
[233048.368766] [    25 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
(priority: 120), next_pid: 22543 (priority: 69), on_cpu: 1,
```

From the worst case latency trace, a `sched_wakeup` is immediately followed by a `sched_switch`.

This is the worst case response time trace to next sleep:

```
[233048.405503] [     0 µs] Event: sched_wakeup, pid: 22543, wake_cpu: 1
[233048.415018] [     7 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
```

```
(priority: 120), next_pid: 22543 (priority: 69), on_cpu: 1,
[233048.431315] [  1059 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 22543
(priority: 69), next_pid: 0 (priority: 120), on_cpu: 1,
```

Response time to first voluntary sleep has around a 1ms response time. This indicates the event loop roughly would wake, do around 1ms of computation and go back to sleep. From our busy loop, this is as expected, and doesn't immediately indicate any issue.

Response time from wake to relief sleep (which is a manual sleep call) is not set. This is as expected since a sleep call is never triggered and an interval will never be generated.

Assume that we did detect an issue relating to this task having delay issues. From our trace, we can eliminate the possibility of a contention issue between different programs by the healthy latency. We can also eliminate the possibility of an IRQ storm, since as stated in case study 2, sustained IRQ activity would generally manifest as softirq thread execution preempting the task.

Therefore it is likely to be a behaviour inside of our event driven task that leads to delay issues.

As seen in this case study, event-driven tasks deviate from the periodic execution model assumed by the response-relief metric, rendering that statistic inapplicable. However, latency and response time measurements remain valuable for eliminating scheduler level interference and IRQ related causes. Even when full diagnosis is not possible, the trace evidence significantly narrows the scope of investigation, pointing toward internal task behavior rather than system level contention.

6.4 Scenario 4: Priority Inversion

We now intentionally simulate a faulty priority inversion workload to demonstrate where the tool could be useful in such a workflow.

While the faulty priority inversion workload manifests as a prolonged blocking similar to Case Study 2, the underlying cause is distinct: priority inversion prevents the lock holder from making progress.

We set up a main high priority control loop (H) in our task, along with medium (M) and low priority (L) periodic threads. There is a global shared mutex that H would attempt to acquire each iteration. L would also attempt to acquire this mutex each iteration, which interferes with H. Note that H is the pid being tracked.

```
taskset -c 2 chrt -f 90 ./priority_inv_broken_bin   // first experiment

taskset -c 2 chrt -f 90 ./priority_inv_work_bin   // second experiment
```

The mutex H is a `pthread_mutex_t` configured without priority inheritance in the first experiment, and with `PTHREAD_PRIO_INHERIT` in the second. The faulty behaviour therefore can be expected in the first experiment.

This is the priority inversion case:

```
[526992.042133] jerry_rt_module: PID=26491, Name=priority_inv_br, minLat=852, maxLat=6296,
[526992.042133] minResponseTimeVoluntarySleepAllTypes=5797,
maxResponseTimeVoluntarySleepAllTypes=1020611,
[526992.042133] minResponseTimeVoluntarySleepReliefBased=1005278,
maxResponseTimeVoluntarySleepReliefBased=11024296
```

We see that the most problematic area is response time relief with about a WC of 11ms. Therefore we investigate its trace.

```
[526992.191259] WORST RESPONSE RELIEF TIME TRACE:
[526992.197595] [     0 µs] Event: sched_wakeup, pid: 26491, wake_cpu: 2
[526992.205896] [     1 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 26492
(priority: 89), next_pid: 26491 (priority: 9), on_cpu: 2,
[526992.221195] [  1008 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 26491
(priority: 9), next_pid: 26493 (priority: 49), on_cpu: 2,
[526992.236609] [ 11018 µs] Event: sched_switch, preemption: 1, voluntary: 0, prev_pid: 26492
(priority: 89), next_pid: 26491 (priority: 9), on_cpu: 2,
[526992.251868] [ 11022 µs] Event: sys_sleep, pid: 26491
[526992.258800] [ 11024 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 26491
(priority: 9), next_pid: 26492 (priority: 89), on_cpu: 2,
```

The amount of response relief delay is very large with the majority of the time being spent during a control iteration where H gets switched out, and switched back in ∼10000µs later. Remark that response time relief tracks a single control loop iteration.

From the trace we can observe that during iteration, it voluntarily switches itself out. This is likely due to a blocking operation, as there are no preempting processes and/or IRQ thread activity. Given the only blocking primitive in the region is a mutex, the evidence strongly suggests that lock contention as the cause. With further investigation in the code itself, one can realize that L holds the lock while being preempted by M, causing the lock to not be released until L gets scheduled again.

One way of dealing with it is likely removing it from the system entirely (case study 2). Another way of dealing with it can be to convert the mutex to use priority inheritance. With priority inheritance enabled, the scheduler temporarily boosts the priority of the lock holding task (L), allowing it to run ahead of M and release the mutex as soon as possible.

This is the priority inheritance mutex implemented case:

```
[527339.798826] minResponseTimeVoluntarySleepReliefBased=1006278,
maxResponseTimeVoluntarySleepReliefBased=1897426
[527339.970980] WORST RESPONSE RELIEF TIME TRACE:
[527339.977380] [     0 µs] Event: sched_wakeup, pid: 26607, wake_cpu: 2
[527339.985723] [     1 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 26608
(priority: 89), next_pid: 26607 (priority: 9), on_cpu: 2,
[527340.001082] [  1008 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 26607
(priority: 9), next_pid: 26608 (priority: 9), on_cpu: 2,
[527340.016310] [  1891 µs] Event: sched_switch, preemption: 1, voluntary: 0, prev_pid: 26608
(priority: 89), next_pid: 26607 (priority: 9), on_cpu: 2,
[527340.031739] [  1895 µs] Event: sys_sleep, pid: 26607
[527340.038679] [  1897 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 26607
(priority: 9), next_pid: 26609 (priority: 49), on_cpu: 2,
```

The PI mutex dramatically reduces the response relief delay, confirming that the long stalls observed in the original trace were caused by priority inversion rather than scheduler contention or IRQ interference.

While the resulting ∼1ms worst case handling delay may still be unacceptable for a hard real time control loop, the trace evidence allowed us to accurately identify the root cause, apply a targeted fix, and validate its effectiveness. This demonstrates how the tool could help with iterative diagnosis and design refinement beyond one shot performance checks.

6.5 Scenario 5: IRQ Demonstration

This case study demonstrates that the tool can distinguish between scheduler latency, IRQ scheduling latency, and handler execution time, and can localize delay to the IRQ handler body itself. We focus on threaded IRQ handlers response time (used in `PREEMPT_RT` to run soft handlers).

To make precise tracking[1] possible, I wrote a new kernel patch to set up threaded IRQ tracepoints in the kernel. To actually test an IRQ line, I hooked up a pushup button on a breadboard with the Pi, toggling GPIO pin 17's connection to the 3.3V current. Then, I made a kernel module (`irq_button_module`) to register the new hardirq and threaded IRQ handler functions with pin 17.

We intentionally lengthen the time that an IRQ handler function runs to validate if our tool can catch it. This is done by adding a `pr_info` call into the threaded IRQ handler. `pr_info` can contend on printk/logging paths and triggers console flushing, so it's both expensive and jittery inside interrupt context.

This is the trace from pulling the IRQ line a few times:

```
[565189.237676] IRQ irq/42-buttoncl (PID=3264):
[565189.244321]   service_time_min/max (wake->first sleep): 7497593 / 8304611 ns
[565189.253818]   latency_min/max (wake->first run): 8741 / 11944 ns
[565189.262222]   per interrupt bottom half handling time (single iteration): 2167 / 8276056 ns
```

We can clearly see that the latency metrics look healthy with worst case 11µs, indicating the IRQ thread is woken and scheduled promptly. While the most critical metrics are service time (response time) and irq per handler time with worst cases of $\geq$ 8000µs.

Therefore we could take a look at the respective traces.

```
[565189.326447] WORST RESPONSE TIME TRACE:
[565189.332668] [     0 µs] Event: sched_wakeup, pid: 3264, wake_cpu: 0
[565189.341394] [    12 µs] Event: sched_switch, preemption: 0, voluntary: 0, prev_pid: 0
(priority: 120), next_pid: 3264 (priority: 49), on_cpu: 0,
[565189.356975] [  8304 µs] Event: sched_switch, preemption: 0, voluntary: 1, prev_pid: 3264
(priority: 49), next_pid: 0 (priority: 120), on_cpu: 0,

[565604.353195] WORST IRQ HANDLING TIME TRACE:
[565604.359206] [     0 µs] Event: irqt_entry, pid: 3264
[565604.366078] [  8304 µs] Event: irqt_exit, pid: 3264
```

The response time trace indicates that once the IRQ thread is scheduled in, it continuously runs for the entire duration. IRQ handler trace suggests a similar picture, with handler entry and exit being the only events while handling. This rules out the possibilities of other processes and external IRQ thread interference. This strongly suggests a slow path inside the threaded handler, which code inspection confirms to be `pr_info`.

These are the statistics after removing the `pr_info` call:

```
[564521.988094] IRQ irq/42-buttoncl (PID=11521):
```

[1]No tracepoints existed for IRQ thread handling time, and manual correlation between tracepoints would be unreliable and too complex.

```
[564521.995662]   service_time_min/max (wake->first sleep): 11944 / 30389 ns
[564522.005605]   latency_min/max (wake->first run): 4315 / 16963 ns
[564522.014811]   per interrupt bottom half handling time (single iteration): 1759 / 4074 ns
```

Service time and interrupt time becomes relatively healthy, ∼30µs and ∼4µs respectively.

In a `PREEMPT_RT` system, slow threaded IRQs execute in process context and directly compete with real time tasks for CPU time. Additionally, many control loops rely on IRQ lines to deliver sensor data; delayed IRQ handling therefore translates directly into delayed control decisions. Having dedicated tracepoints and trace handlers for threaded IRQs helps identify and eliminate this failure mode.

7 SMP scenarios and Mitigations

This section highlights a suite of potential concerns regarding such a design. I explain measures implemented to mitigate or reasons such a case is acceptable in this tool.

7.1 Ordering/Visibility across cores for max/min

The original design shares max/min variables for a particular task along different cores of the Pi 4. While tracing, this could result in a new max store, however, this maximum value has not achieved global visibility. Then, on another core, the task would complete another cycle of the metric to now set max. Instead of checking new potential max against the previous max, it would check against the one before the previous. This causes us to potentially lose an update.

Let a Core 1 store into max/min (C_{1s}) at time t_0. Let Core 2's load (C_{2l}) happen at time t_1. This is a "RAW" dependency: C_{1s} needs global visibility before C_{2l}.

Memory barriers are meant for ordering rather than enforcing global visibility across cores. Define global visibility to be when a store on one core is visible from any other core (i.e. When a query on any core to the particular memory location produces the latest store). A `smp_acquire` and `smp_release` pair on max/min, would guarantee ordering rather than visibility.

Suppose we put a acquire/release pair on max and Core 1 writes (C_{1s}) and Core 2 reads (C_{2l}), in this chronological order. When Core 1's new max value is globally visible and core 2 sees it, the memory barrier on max enforces that all the previous memory operations must also be now globally visible.

$$C_{1s} \text{ is globally visible} \implies C_{2l} \text{ will see } C_{1s} \text{ and load/stores before it}$$

This entailment depends on global visibility, therefore it doesn't particularly help us in this scenario as we simply want to view the latest max.

A (Linux spinlock) lock would work as a solution. A lock enforces that writes done before we unlock are guaranteed to be visible to the next thread that later acquires that same lock; a pairwise visibility.

If we put a lock on both max/min, then C_{2l} would be loading the latest value by definition of the lock acquisition.

However, as we are updating max/min on hot paths, using locks is a somewhat heavier solution. We use a CAS (compare and swap) loop instead.

Every time we update max or min for a particular variable, we'd call `cas_update_max_s64` or `cas_update_min_s64`.

```
static __always_inline bool cas_update_max_s64(s64 *p, s64 v)
{
 s64 old = READ_ONCE(*p);

 while (v > old) {
  s64 prev = cmpxchg64(p, old, v);
  if (prev == old) {
   return true;
  }
  old = prev;
 }
 return false;
}
```

`cmpxchg64(p, old, new)` is an atomic RMW (Read Modify Write operation). If it succeeds, the new value becomes the coherently ordered value for that cache line, and other core's subsequent loads will observe that value. The Pi 4 supports 64 bit RMW operations.

A RAW interaction between Core 1 and Core 2 happens where Core 1 is storing a higher value (100) and Core 2 is storing a lower value (95), with max=90 initially. We want to show that this implementation allows for the correct final max=100.

Case 1: No overlap

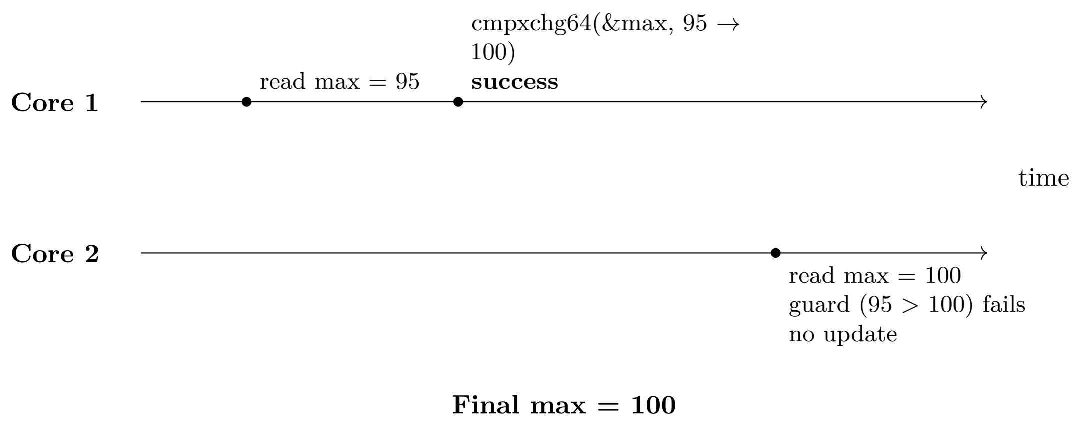

Figure 10: Case 1 (no overlap)

From figure 1, the effect of the cmpxchg64 allows Core 2 load the correct value.

Case 2: Overlap of operations in time

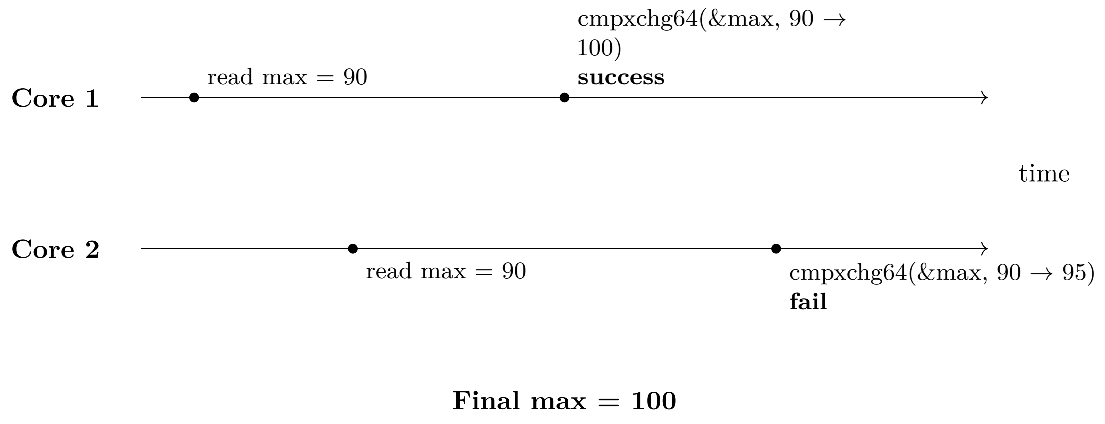

Figure 11: Case 2 (With overlap in operation)

Similarly in case 2, we see that cmpxchg64 allows atomic propagation of the write to the memory location. This allows Core 2 to discover Core 1's update, and realize that it shouldn't update the existing max.

7.2 Trace Update racing (This doesn't solve the fundamental problem)

Now, the max/min values are updated correctly, the trace needs a synchronization mechanism. If not implemented, this may lead to the trace updates racing and becoming non-sensical, as illustrated below.

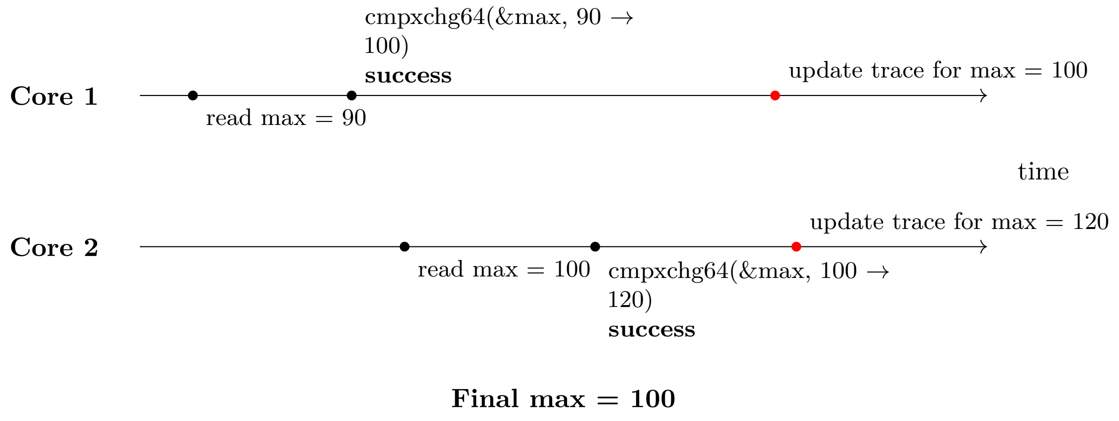
cmpxchg64(&max, 90 → 100)
success
update trace for max = 100
Core 1
read max = 90
time
update trace for max = 120
Core 2
read max = 100
cmpxchg64(&max, 100 → 120)
success
Final max = 100

Figure 12: Trace updates for core 1 and core 2 races (updates in max=100 could corrupt max=120)

We serialize trace writes with a per metric lock, so only one trace update can modify the stored "worst case trace" at a time.

However, a trace update may continue executing if a newer maximum is installed while it is happening. To reduce the likelihood of committing a out of date trace, we recheck the current maximum under the same trace lock before starting collection: trace collection for a value v_0 is initiated only if `max ==` v_0 at lock acquisition time; otherwise, the trace update is skipped.

Consequently, the stored trace always corresponds to the latest (currently visible) maximum: either the earlier trace completes and remains valid (no newer max occurred), or it is skipped and a later update for the newer max is the one that can commit.

During pid removal/module exit, we synchronize so no inflight trace updates can still be mutating trace buffers. After all relevant probes are unregistered, `tracepoint_synchronize_unregister()` waits for any in-flight executions of those probe callbacks to complete. Therefore, after it returns, no CPU can still be executing the trace update critical section inside those callbacks, and any final max triggered trace update that had started running before `tracepoint_synchronize_unregister()` will have finished. For `stop_recording_pid`, we acquire tracelocks to allow for incomplete trace updates to finish before printing. If a `stop_recording_pid` call happens before trace lock is acquired but it had still started handler, our mechanism will not guarantee including this trace.

Trace updates have two guarantees: (1) trace buffer update mechanisms are serialized and do not race, and (2) trace collection is initiated only if the observed maximum still matches the candidate value at trace lock acquisition time.. Occasional loss of an exact trace is acceptable in exchange for minimizing interference with original workload.

7.3 Atomic counter violation

Violation counters (i.e. `latency_counter`) could be updated concurrently on multiple cores. If we implement this as a plain integer with separate load and store operations (e.g. `READ_ONCE(x) + 1; WRITE_ONCE(x)`), increments can race and be lost due to the non-atomic RMW sequence.

Therefore, we use `atomic_t` per metric violation counts, ensuring that increments are performed atomically and no updates are lost under concurrent execution.

Note that `atomic_t` is solely used to guarantee atomicity of increments. No additional synchronization with other metric data is implied or required.

7.4 Start index updates not locked

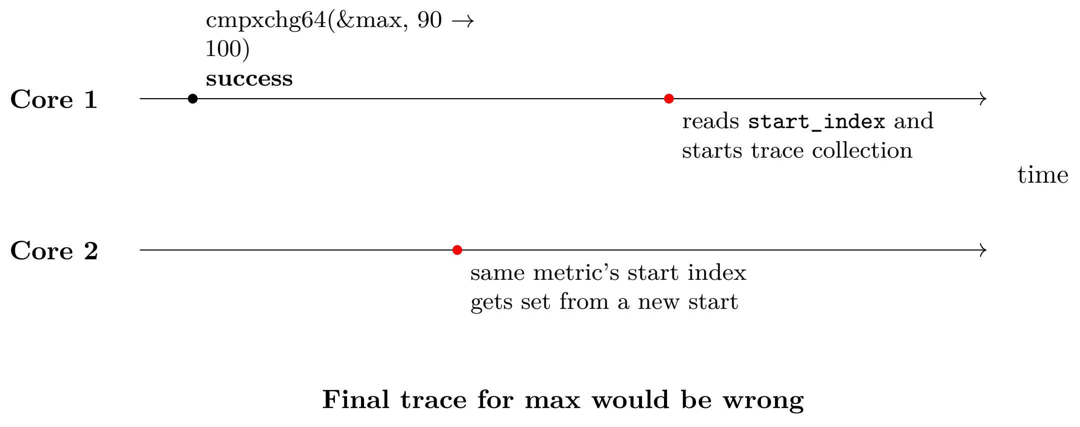
cmpxchg64(&max, 90 → 100)
success
Core 1
reads start_index and starts trace collection
time
Core 2
same metric's start index gets set from a new start
Final trace for max would be wrong

Figure 13: Wrong trace reference sequence

The start index is intentionally published without a lock to avoid adding contention to the tracepoint hot path, as it is updated on every cycle start. Because of this, it is possible for `start_index` to be updated by a newer cycle before trace collection for an earlier max begins. In that case, the trace collector may observe the start index of a later cycle, losing the original trace (Figure 4).

The tool's primary correctness goal is the numeric max/min values and violation counts, which are protected by CAS loops to ensure correctness even in races. The stored trace, on the other hand, is a best effort diagnostic rather than a strict reconstruction guarantee.

Under very high event rates, the fault described above is very likely. In this case, the trace collection observe a wakeup that occurs after the cycle which produced the max. Since a cycle start must precede its corresponding cycle end, such a trace would be an invalid and discarded by `collect_slo_trace_common`.

8 Future Additions (notes)

Evaluations on a `PREEMPT_RT` kernel.

Potential races exist that I haven't caught (E.g. `latency_last_wakeup` haven't finished updating, and the `sched_switch` in happens on another core. This is a tracepoint on core 1 hasn't finished executing and there is the end of the cycle happening on Core 2. However, this fault I don't particularly value as this is likely a failure to record a minimum, rather than a failure to record a worst case.)

To allow the use of tgids rather than pids, as there isn't support for threads spawned under a pid yet.

To allow for easier IRQ setup without doing a kernel patch.

Finding a way to use less memory when running the module for more strict embedded environments.