Make RCU do less (& later)!

Presenters:
- Joel Fernandes (Google)
- Uladzislau Rezki (Sony)
- Rushikesh Kadam (Intel)

Intel power data courtesy: Sitanshu Nanavati.
Overview

● Discuss what RCU does at high-level (not how it works!).

● Discuss the 2 main issues we found:
  ○ On a mostly idle system, RCU activity can disturb the idleness.
    ■ RCU default config required to keep tick on when idle and CBs queued.
    ■ RCU constantly asked to queue callbacks on a lightly loaded system.

● Discuss possible solutions.

● Taking questions in the end as time permits (and then hallway)
What RCU does?

Data-Structure read/modification usecase:

- RCU reader critical section protected by "read lock"
- RCU writer critical section protected by regular locks.
- Reader and writer execute concurrently.
- Writer creates copy of obj, writes to it and switches object pointer to new one (release ordered write).
- Writer Garbage Collects old object after waiting (update)
What RCU does?

- That’s just one use case, there are many uses of RCU.

All use cases need same basic tools:

- Lock-less markers of a critical section (CS). Call it “reader”.
- Start waiting at some point in time \((t = T_0)\).
- Stop waiting after all readers that existed at \(T_0\) exited CS \((t = T_1)\).

Note: Start \((T_0)\) is when a “callback is queued”, Stop is when a “callback is invoked”.
What RCU does?

- On a local CPU (running in **kernel mode** with CB queued).

  Upper red arrows are timer tick checking are there readers left? If not, report.

  Lower red arrows are timer tick: have ALL CPUs reported? If yes, invoke CB. If no, try again.

Queued a Callback (CB)
What RCU does?

- On a local CPU (running in **idle mode** with CB queued).

  Upper red arrows are timer tick checking are there readers left? If not, report.
  **THESE NOT NEEDED - AS CPU CANNOT BE IN RCU READER CRITICAL SECTION!**

  Lower red arrows are timer tick: have ALL CPUs reported? If yes, invoke CB. If no, try again.
  **THESE STILL NEEDED - AS local CPU has queued CB.**

Queued a Callback (CB)
And soon go idle...
What RCU does?

- You see the problem?
  - RCU can block the timer tick from getting turned off!
  - Negates power-savings of CONFIG_NOHZ_IDLE

(To be fair to RCU, this workload queues a lot of RCU Callbacks on otherwise idle CPUs, requiring RCU to do work on these otherwise-idle CPUs).
Issue 1: RCU keeping the scheduler tick ON when idle.

- “Local Video Playback” use-case has 2500+ timer wakes per second. A large chunk of the wakes result from constantly queued RCU callbacks.

- RCU wakes are seen at HZ rate (red boxes) between graphics 16.6ms activity (blue boxes)

- Blocks deeper Package C-states. Impacts power
How bad are idle ticks for power

- We know idle ticks are bad for power, but we did not know how bad!

- In Video playback, timer wakes amount to < 2% CPU load, but blocked deepest package C-states (PC8) for 25+% of the time, causing 10+% in SoC + memory power.

- If you are profiling CPU load, then you will likely miss the impact of wakes (use powertop)
Why are idle ticks so bad for power?
What are package C-states

- Traditionally ACPI C-states are **CPU** low power states
- Idle governor picks C-states based on OS next event prediction and C-states exit latency & target residency
- CPU C-states have low exit latency & target residency, so doesn’t block much on ticks

- **System-On-Chip** architecture provides opportunity to extend the OS C-states hints to power manage the entire SoC.
- SoCs have power management unit which coordinates CPU, IP blocks and common logic, to put entire SoC in low power
- Package C-states add extended C-states with higher exit latency & longer power breakup time.

```c
static struct cpuidle_state adl_cstates[] __initdata = {
    {
        .name = "C1",
        .exit_latency = 1,
        .target_residency = 1,
    },
    {
        .name = "C1E",
        .exit_latency = 2,
        .target_residency = 4,
    },
    {
        .name = "C6",
        .exit_latency = 220,
        .target_residency = 600,
    },
    {
        .name = "C8",
        .exit_latency = 280,
        .target_residency = 800,
    },
    {
        .name = "C10",
        .exit_latency = 680,
        .target_residency = 2000,
    },
    {
        .enter = NULL }
};
```
But why does some RCU configs keep tick on if so bad for cpuidle?

This is required in default RCU configurations as CBs are invoked on same CPU they were queued on, in a softirq.

Advantages:

- Executing CBs on queuing CPU eliminates need for CB list locking.
- No need for additional thread wake ups as local softirq execs CB.
- Cache-line is likely hot from queuing and CB would not incur misses.

These can be especially useful on busy systems and large #CPU server!

- Periodic tick check helps with timely detection of GP end and thus CB exec.
Issue 1: RCU keeping the scheduler tick ON when idle.

Solution for newer kernels: CONFIG_RCU_N0CB_CPU (Execute RCU CBs in per-cpu threads.)
Issue 1: RCU keeping the scheduler tick ON when idle.

Solution for newer kernels: CONFIG_RCU_NOCB_CPU

Can cause performance overhead on system with frequent CB queue/exec!
Issue 1: RCU keeping the scheduler tick ON when idle.

Solution for newer kernels: CONFIG_RCU_NOCB_CPU

However, can be great for power and CPU isolation…

- Scheduler may move threads to non-idle CPUs thus leaving more idle.
- **Both** starting of new grace periods, and executing CBs are moved out of the softirq context and into threads.
CONFIG_RCU_NOCB_CPU saves lots of power

- RCU callback offload unblocks dynticks-idle and hence reduces timer wakes.
- RCU callback offload does increase the scheduler wakes marginally, but reduces total platform wakes.
- Improves Package C-states residency and hence SoC + Memory power.

Use-case: Local video playback via Chrome browser, VP9 1080p @ 30 fps content

Device: Chrome reference device, AlderLake Hybrid CPU with 2 Cores (with Hyperthreading) + 8 Atoms
New option: CONFIG_RCU_NOCB_CPU_ALL

- If you enable CONFIG_RCU_NOCB_CPU, you still need to specify rcu_nocbs=0-N to make it work.

So...

- New option CONFIG_RCU_NOCB_CPU_ALL was added to just enable nocb for all CPUs by default.
Can we do even better?

Observations:

- When a system is mostly idle, most CBs don’t need to execute right away, some can be delayed as long as needed!

- Some CBs in the system “trickle” frequently.
Observation: ChromeOS when idle

- Some CBs in the system “trickle” frequently.
- Several callbacks constantly queued.

rcutop refreshing every 5 seconds. ChromeOS logged in with screen off. Device on battery power.

<table>
<thead>
<tr>
<th>Callback</th>
<th>Queued</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>inode_free_by_rcu</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>delayed_put_task_struct</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>k_itimer_rcu_free</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>radix_tree_node_rcu_free</td>
<td>16</td>
<td>27</td>
</tr>
<tr>
<td>rcu_work_rcufn</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>put_cred_rcu</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>delayed_put_pid</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>unbind_fence_free_rcu</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>dst_destroy_rcu</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>__i915_gem_free_object_rcu</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>thread_stack_free_rcu</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
Observation: ChromeOS
Display pipeline

Display pipeline in ChromeOS constantly opens/close graphics buffers.
Observation: Logging in Android (as example)

Android uses CONFIG_RCU_NO_CB by default to offload all CPUs.
Observation: Logging in Android (as example)

Example: Logging during static image (Android).

Static image is important use-case for power testing on Android. The system is mostly idle to minimize a power drain of the platform:

- CPU stops refreshing panel and panel self-refreshes on it own.
- CPUs spend most of their time in deepest C-state
- SoC bandwidth is minimal (memory bus, CPU/cache frequencies, etc.).

Logging does constant file open/close giving RCU work when FDs get freed. As a side effect of such periodic light load, many wakeups happen due to frequent kicking an RCU-core for initializing a GP to invoke callbacks after it passes.
Observation: Logging in Android (as example)

Below is a post process of scheduler ftrace for static image use-case during 30 seconds

(this is with CONFIG_RCU_NOCB_CPU with all CPUs offloaded).

<table>
<thead>
<tr>
<th>Event</th>
<th>PID</th>
<th>State</th>
<th>Interval (min)</th>
<th>Interval (max)</th>
<th>Interval (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcuop/2</td>
<td>33</td>
<td>woken-up</td>
<td>1320</td>
<td>71837</td>
<td>9807</td>
</tr>
<tr>
<td>rcuop/3</td>
<td>40</td>
<td>woken-up</td>
<td>1582</td>
<td>78649</td>
<td>9744</td>
</tr>
<tr>
<td>rcuop/0</td>
<td>15</td>
<td>woken-up</td>
<td>1520</td>
<td>80442</td>
<td>8873</td>
</tr>
<tr>
<td>rcuop/1</td>
<td>26</td>
<td>woken-up</td>
<td>1414</td>
<td>80043</td>
<td>8846</td>
</tr>
<tr>
<td>rcuop/0</td>
<td>14</td>
<td>woken-up</td>
<td>73</td>
<td>27855</td>
<td>6217</td>
</tr>
<tr>
<td>idd@1.0</td>
<td>1116</td>
<td>woken-up</td>
<td>231</td>
<td>17442186</td>
<td>4005</td>
</tr>
<tr>
<td>rcu_preempt</td>
<td>13</td>
<td>woken-up</td>
<td>39</td>
<td>8505</td>
<td>3991</td>
</tr>
<tr>
<td>iddd</td>
<td>1195</td>
<td>woken-up</td>
<td>92</td>
<td>16375</td>
<td>1437</td>
</tr>
</tbody>
</table>

A trace was taken on the ARM big.LITTLE system. It is obvious that the biggest part belongs to the “idd logger” whereas a second place is fully owned by the RCU-core subsystem marked as red.
Observation: Logging in Android (as example)

RCU mostly invokes callbacks related to the VFS, SELinux subsystems during logging:

- `file_free_rcu()`
- `inode_free_by_rcu()`
- `i_callback()`
- `__d_free()`
- `avc_node_free()`

Since system is lightly loaded and a number of posted callbacks to be invoked are rather small, between 1-10, such pattern produce most of the wakeups (in static image use-case) to offload a CPU with __only__ few callbacks there.
Observation: Logging in Android

6-7 milliseconds interval

Only a few callbacks are invoked
Issue 2: RCU queuing CBs on lightly loaded system

Let us explore some solutions to this…
Issue 2: RCU queuing CBs on lightly loaded system

Solution 1: Delay RCU processing using jiffies_till_{first,next}_fqs

- Great power savings

<table>
<thead>
<tr>
<th>jiffies_till_first_fqs &amp; jiffies_till_next_fqs</th>
<th>Baseline (NOCB)</th>
<th>= 8, 8</th>
<th>= 16, 16</th>
<th>= 24, 24</th>
<th>= 32, 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoC+Memory, power savings w.r.t Baseline</td>
<td>0%</td>
<td>2%</td>
<td>3%</td>
<td>3.4%</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

- Problem:
  - Causes slow down in ALL call_rcu() users globally whether they like it or not.
  - Causes slow down in synchronize_rcu() users globally.
  - Significantly regresses boot time.
Issue 2: RCU queuing CBs on lightly loaded system

Solution 1: Jiffies causes massive synchronize_rcu() slowdown.

- ChromeOS tab switching autotest
  - Due to synchronize_rcu() latency increases quickly from 23 ms to 169 ms (with changing jiffies from 3 to 32)

- The same evaluation with synchronize_rcu_expedited() gives us a latency of < 1 msec at jiffies = 32
Solution 1: Jiffies increase causing function tracer issues

Several paths in ftrace code uses synchronize_rcu():

For but 2 examples:

- `pid_write()` triggered by write to `/sys/kernel/tracing/debug/tracing/set_ftrace_pid`
- `ring_buffer_code` such as `ring_buffer_resize()`

End result is `trace-cmd record -p function_graph` can take several more seconds to start and stop recording, than it would otherwise.
Issue 2: RCU queuing CBs on lightly loaded system
Solution 1: Jiffies causing boot-time issues (SELinux)

SELinux enforcing during ChromeOS boot up invokes synchronize_rcu()

[ 17.715904] => __wait_rcu_gp
[ 17.715904] => synchronize_rcu
[ 17.715904] => selinux_netcache_avc_callback
[ 17.715904] => avc_ss_reset
[ 17.715904] => sel_write_enforce
[ 17.715904] => vfs_write
[ 17.715904] => ksys_write
[ 17.715904] => do_syscall_64
Issue 2: RCU queuing CBs on lightly loaded system
Solution 1: Jiffies causing per-cpu refcount regression

- RCU used to toggle atomic-mode and vice versa
- Can badly hurt paths that don’t really want to free memory but use call_rcu() for some other purposes. Like suspend.
- call_rcu() slow down affects percpu refcounters
- These counters use RCU when switching to atomic-mode
  - __percpu_ref_switch_mode() -> percpu_ref_switch_to_atomic_sync().
- This call slows down for the per-cpu refcount users such as blk_pre_runtime_suspend().

This is why, we cannot assume call_rcu() users will mostly just want to free memory. There could be cases just like this, and blanket slow down of call_rcu() might bite unexpectedly.
Solution 1: Jiffies with expedited option

- The previous synchronize_rcu() issues can be mitigated by using expedited boot option which expedites while ensuring good power efficiency.

- However, experiments showed that using expedited RCU with jiffies, still causes a boot time regression.

- Also, the expedited option is expensive, and can affect real-time workloads.
Issue 2: RCU queuing CBs on lightly loaded system

Solution 2: Delay RCU CB processing (Lazy RCU)

- Delay Callback execution as long as possible.
- Introduce new API for lazy-RCU (call_rcu_lazy).
- Need to handle several side-effects:
  - RCU barrier.
  - CPU hotplug etc.
  - Memory pressure.
  - Offloading and De-offloading.
Issue 2: RCU queuing CBs on lightly loaded system
Solution 2: Delay RCU CB processing (Lazy RCU)
Issue 2: RCU queuing CBs on lightly loaded system

Solution 2: Delay RCU CB processing (Lazy RCU)
Issue 2: RCU queuing CBs on lightly loaded system

Lazy RCU: design approach

Can cause performance overhead on system with frequent CB queue/invoke due to locking!
Issue 2: RCU queuing CBs on lightly loaded system
Lazy RCU: design approach - re-use the bypass list.

By-pass list is per-cpu and but batches CBs and wakes RCU 2 jiffy delay, and relieves lock contention on the main CB list.
Issue 2: RCU queuing CBs on lightly loaded system

Lazy RCU: design approach - re-use the bypass list.

Flush the bypass list if there is memory pressure, or lengthy timer expires!
Issue 2: RCU queuing CBs on lightly loaded system
Solution 2: Delay RCU CB processing (Lazy RCU)

RCU lazy further reduces 300+ wakes per seconds, and improves SoC package C-states residency & Power

Use-case: Local video playback via Chrome browser, VP9 1080p @ 30 fps content

Device: Chrome reference device, AlderLake Hybrid CPU with 2 Cores (with Hyperthreading) + 8 Atoms
Issue 2: RCU queuing CBs on lightly loaded system

Solution 2: Delay RCU CB processing (Lazy RCU)

Latest Patches:

https://lore.kernel.org/all/20220819204857.3066329-1-joel@joelfernandes.org/

Summary:

- Introduce new API for lazy-RCU (call_rcu_lazy).
- Queue CBs into the Bypass list.
- Flush the Bypass list when:
  - Non-Lazy CBs show up.
  - Bypass list grows too big or is too old.
  - Memory is low.
- Several corner cases now handled (rcu_barrier, CPU hotplug etc).
Home screen swipe (as example)
Home screen swipe power (~3% delta)
Some CBs in the system “trickle” frequently.

Several callbacks constantly queued.

rcutop refreshing every 5 seconds. ChromeOS logged in with screen off. Device on battery power.
Drawbacks and considerations

- Depends on user of `call_rcu()` using lazy
  - If a new user of `call_rcu()` shows up, it would go unnoticed and negate the benefits.
  - Updates to docs may help: https://docs.kernel.org/RCU/whatisRCU.html#id11

- Risk of user using `call_rcu_lazy()` accidentally when they should really use `call_rcu()`. For example, a use case requiring synchronous wait.

- Risks on memory pressure:
  - Protection is enough on extreme condition?
  - Can test with more test cases such as ChromeOS memory pressure test.
Thanks!

- Paul McKenney (for putting up with us).
- Presenters.
- LPC sponsors and organizers.
- Frederic Weisbec for reviewing code.

Questions?