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 blocking the scheduler tick ON when idle.
    ■ RCU constantly queuing callbacks on a lightly loaded system.
● Discuss possible solutions.
What RCU does?

- 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 GCs 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).
- Start waiting at some point in time \( t = T_0 \).
- Stop waiting until all readers that existed at \( T_0 \) exited CS.
What RCU does?

- On a local CPU (running in kernel mode).

- 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, execute CB. If no, try again.

- Queued a Callback (CB)
What RCU does?

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

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, execute CB. If no, try again.
**THESE STILL NEEDED - AS local CPU has queued CB.**
What RCU does?

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

- This happens even in user mode
- NOHZ_FULL systems typically turn tick off. RCU can keep it on.
Issue 1: RCU keeping the scheduler tick ON when idle.

- “Local Video Playback” use-case has 2500+ wakes per second. A large chuck of the wakes result from RCU callbacks blocking the dynticks-idle mode

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

- Blocks deeper Package C-states. Impacts power
Why was RCU keeping the tick on?

- By default RCU executes callbacks on the same CPU that queued them, in a softirq.

- If there are Callbacks queued, keeping the tick on ensures the CBs are executed in a timely fashion.
Issue 1: RCU keeping the scheduler tick ON when idle.

Possible solution: Using `CONFIG_FAST_NOHZ` option

- This option permits CPUs to enter the dyntick-idle state (the state where the tick is turned off) even if they have CBs queued.

- Idle CPUs with callbacks are kept idle for a minimum number of jiffies before rechecking of the RCU state.
Issue 1: RCU keeping the scheduler tick ON when idle.

Solution for newer kernels:

- **CONFIG_FAST_NO_HZ** is removed in recent kernels.
- **CONFIG_RCU_NOCB_CPU**: Execute RCU CBs in per-cpu threads.
- Scheduler may or may not move threads to non-idle CPUs and is in control of whether CPU needs to be idle or execute callbacks.
- **Both** starting of new grace periods, and executing CBs are moved out of the softirq context and into threads.
Issue 1: RCU keeping the scheduler tick ON when idle.

- RCU callback offload unblocks dynticks-idle and hence reduces timer wakes.
- RCU callback offload does increase scheduler wakes marginally, but reduces total platform wakes.
- Improves Package C-states residency 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
Observation: ChromeOS when idle

- Several callbacks constantly queued.
- ChromeOS login + screenoff
- Device is on battery.
**Observation:**

**ChromeOS Display pipeline**

Display pipeline in ChromeOS constantly opens/close graphics buffers.
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:

- Panel refresh-rate is zero, i.e. it is stopped and power collapsed
- 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 inducing RCU pressure 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 wakeup trace log of static image use-case during 30 seconds.

A trace was taken on the ARM big.LITTLE system. It is obvious that the biggest part belongs to the “iddd 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
Home screen swipe (as example)

default

cpu_lazy_v6
Home screen swipe power (~3% delta)

avg: ~365 mA

avg: ~352 mA
Issue 2: RCU queuing CBs on lightly loaded system

- Observation: When a system is relatively idle, most CBs in the system don’t need to be executed soon, we can in fact delay them as long as needed.

- Selectively identify frequently occurring CBs in the system that “trickle”.
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>= 3,3 (default)</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>11.1%</td>
<td>13.1%</td>
<td>13.9%</td>
<td>14.4%</td>
<td>14.2%</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
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.
Issue 2: RCU queuing CBs on lightly loaded system
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)

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.
  - Memory is low.
- Several corner cases now handled (rcu_barrier, CPU hotplug etc).
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
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)

rcutop confirms callbacks are getting queued but not executed.

```
<table>
<thead>
<tr>
<th>Callback</th>
<th>Queued</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>avc_node_free</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>k_itimer_rcu_free</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>thread_stack_free_rcu</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>file_free_rcu</td>
<td>576</td>
<td>0</td>
</tr>
<tr>
<td>delayed_put_pid</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>radix_tree_node_rcu_free</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>i_callback</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>__d_free</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>dst_destroy_rcu</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>epi_rcu_free</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>delayed_put_task_struct</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>inode_free_by_rcu</td>
<td>94</td>
<td>0</td>
</tr>
</tbody>
</table>
```
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.

● Risk of user using call_rcu_lazy() in a synchronous use case accidentally.

● Risks on memory pressure:
  ○ Protection is enough on extreme condition?

● Helping users choose the right API variant
  ○ Updates to docs may help: https://docs.kernel.org/RCU/whatisRCU.html#id11