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?

- 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). 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.
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.**
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 the main RCU maintainer, this issue is courtesy of the use case queuing a lot of RCU Callbacks on otherwise idle CPUs, in the first place).
What RCU does?

- This happens even in user mode
- NOHZ_FULL systems typically turn tick off.
- RCU can keep it on (if CBs are queued on a ‘nohz full’ CPU)
Issue 1: RCU keeping the scheduler tick ON when idle.

- “Local Video Playback” use-case has 2500+ 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, RCU 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 idle ticks are so bad for power

What are package C-states?

- Traditionally ACPI C-states were CPU 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.
- 1000 HZ ticks do not block core C-states much
- E.g. Sandy Bridge C-states table (2011)

```c
static struct cpuidle_state snb_cstates[] __initdata = {
    {
        .name = "C1",
        .exit_latency = 2,
        .target_residency = 2,
    },
    {
        .name = "C1E",
        .exit_latency = 10,
        .target_residency = 20,
    },
    {
        .name = "C3",
        .exit_latency = 80,
        .target_residency = 211,
    },
    {
        .name = "C6",
        .exit_latency = 104,
        .target_residency = 345,
    },
    {
        .name = "C7",
        .exit_latency = 109,
        .target_residency = 345,
    },
    {
        .enter = NULL
    }
};
```
Why idle ticks are so bad for power

What are package C-states?

SoC architecture provides opportunity to extend the OS C-states hints to power manage the entire SoC.

SoCs have power management unit (HW + microcode), which coordinates CPU, IP blocks and common logic, to put entire SoC in low power mode.

Package C-states add extended C-states with high exit latency & target residency.

1000 HZ ticks would impact deeper package C-states,

E.g. AlderLake C-state table 2021

```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
};
```
Why was RCU keeping the tick on?

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

Advantages:

- Timely detection of GP end and thus execution of queued CBs.
- 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!
Issue 1: RCU keeping the scheduler tick ON when idle.

Possible solution: Using CONFIG_FAST_NOHZ option

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

- Idle CPUs with callbacks check RCU state every 4 jiffies.
  - 4 jiffies for non-kfree CBs.
  - 6 jiffies or so for kfree CBs.
Issue 1: RCU keeping the scheduler tick ON when idle.

Solution for newer kernels: \texttt{CONFIG_RCU_NOCB_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 its 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).

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
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
Issue 2: RCU queuing CBs on lightly loaded system
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.
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)

- 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)
Solution 2: Delay RCU CB processing (Lazy RCU)

Issue 2: RCU queuing CBs on lightly loaded system

Intro: Life Cycle of a grace period

- Waiting for a new GP request
- Propagate start of GP down the TREE (rcu_gp_init)
- Force Quiescent State (FQS) loop (rcu_gp_qe_loop)
- Are ALL QS marked? (root node qs_mask == 0)
- Mark and Propagate GP end down tree (rcu_gp_cleanup sets gp_seq of rcu_state)
- Mark CPU QS
- Propagate QS up TREE
- Is a GP in progress?
- All CPUs done?
  (Set Root node qs_mask = 0)
- Synchronize RCU
- Queue wake up callback (rcu_seqcallback_queue)
- Request a new GP (rcu_start_is_gp)
- Sleep
- Wake up
- Continue
- Softirq CB exec

DELAYED
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 (almost) lock free!
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.
  - Memory is low.
- Several corner cases now handled (rcu_barrier, CPU hotplug etc).
Home screen swipe (as example)

default

cpu_lazy_v6
Home screen swipe power (~3% delta)

avg: ~365mA

avg: ~352mA
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?