What Happens When 4096 Cores All Do synchronize_rcu_expedited()?
Overview

• What *Should* Happen When 4096 Cores All Do synchronize_rcu_expedited()?
• Overview of Algorithm for synchronize_rcu_expedited()
• Expedited Grace Period Example
• Benchmarking
• Benchmarking on 4096 CPUs
• Summary and Lessons (Re)learned
What *Should* Happen When 4096 Cores All Do `synchronize_rcu_expedited()`?
What Should Not Happen When 4096 Cores All Do synchronize_rcu_expedited()?
What Should Not Happen When 4096 Cores All Do synchronize_rcu_expedited()?
**What Should Not Happen When 4096 Cores All Do synchronize_rcu_expedited()?**

Then What Instead?

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QS 0</td>
<td>QS 0</td>
<td>QS 0</td>
<td>QS 0</td>
</tr>
<tr>
<td></td>
<td>QS 1</td>
<td>QS 1</td>
<td>QS 1</td>
<td>QS 1</td>
</tr>
<tr>
<td></td>
<td>QS 2</td>
<td>QS 2</td>
<td>QS 2</td>
<td>QS 2</td>
</tr>
<tr>
<td></td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td></td>
<td>QS N</td>
<td>QS N</td>
<td>QS N</td>
<td>QS N</td>
</tr>
</tbody>
</table>
RCU Grace Period Properties

- **Grace Period**: Time during which every CPU/task spends some time outside of an RCU read-side critical section
  - Any critical section in progress at the beginning of a grace period must end before that grace period ends
    - RCU read-side critical section spans `rcu_read_lock()` to `rcu_read_unlock()`
    - RCU grace period wait: `synchronize_rcu_expedited()` and friends

- Grace periods are independent of CPU/task requesting them
- A single grace period can serve several requests
- In fact, single non-expedited grace periods often serve thousands of requests in Linux kernels
A grace period can serve multiple updates, decreasing the per-update RCU overhead.
What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()?

Time

CPU 0

QS 0

CPU 1

QS 0

CPU 2

\[\quad\]

CPU N

QS 0

They all should share the same grace period!!!

What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()? (Or Maybe This)

Time

CPU 0
  QS 0
  QS 1

CPU 1
  QS 0
  QS 1

CPU 2
  QS 0
  QS 1

CPU N
  QS 0
  QS 1

Or share two grace periods, depending on timing.
What Else Should Not Happen When 4096 Cores All Do synchronize_rcu_expedited()?
What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?

*No Single Global Locks, Please!!!*
What Should Happen Instead When 4096 Cores All Do synchronize_rcu Expedited()?

Instead, use lots of different locks!!!
Tree RCU's rcu_node Combining Tree to the Rescue!

Separate locks for each instance!!!
What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?
What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu Expedited()?

No Frequently Updated Shared Variables, Please!!!
What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()?

Instead, Use Lots of Shared Variables!!!
Again, Tree RCU's Combining Tree to the Rescue!

Separate variables for each instance!!!
Non-Requirements

- **Real-time response for synchronize_rcu Expedited()**
  - Must wait for readers in any case
  - RCU priority boosting carried out, but more diagnostic than realtime
  - So some variation in timings is to be expected

- **Constant synchronize_rcu_Expedited() latency**
  - After all, synchronize_rcu() latency increases with number of CPUs

- **Big-system performance of synchronize_rcu_Expedited() to the exclusion of all else**
  - Heavy update workloads better served by synchronize_rcu()
Overall synchronize_rcu_expedited() Algorithm
High-Level synchronize_rcu_expedited() Algorithm

- For each non-idle online CPU:
  - Send IPI
  - Handler determines if CPU is in quiescent state (context switch, user-mode execution, idle, cond_resched_rcu_qs(...))
  - If so, report quiescent state
  - If not, set CPU-local state so that next quiescent-state entry is reported

- When all non-idle online CPUs has reported a quiescent state, grace period is complete
High-Level synchronize_rcu Expedited() Algorithm

- For each non-idle online CPU:
  - Send IPI
  - Handler determines if CPU is in quiescent state (context switch, user-mode execution, idle, cond_resched_rcu_qs(...)
  - If so, report quiescent state
  - If not, set CPU-local state so that next quiescent-state entry is reported

- When all non-idle online CPUs has reported a quiescent state, grace period is complete

- The trick is doing this without bottlenecks...
Overall Approach to Concurrent-Code Optimization

- Parallel Access Control
- Work Partitioning
- Resource Partitioning and Replication
- Interacting With Hardware
- Weaken
- Batch
- Partition
Optimize Expedited Grace Periods

- Partition
  - Use the rcu_node combining tree!
Optimize Expedited Grace Periods

- **Partition**
  - Use the rcu_node combining tree!

- **Batch**
  - Need a mechanism to piggyback off others' expedited grace periods
Optimize Expedited Grace Periods

- **Partition**
  - Use the rcu_node combining tree!

- **Batch**
  - Need a mechanism to piggyback off others' expedited grace periods

- **Weaken**
  - My normal advice would be to use RCU, but this *is* RCU...
Optimize Expedited Grace Periods

- **Partition**
  - Use the rcu_node combining tree!

- **Batch**
  - Need a mechanism to piggyback off others' expedited grace periods

- **Weaken**
  - My normal advice would be to use RCU, but this *is* RCU...

- **Hardware**
  - Need to be portable, so no FPGAs or GPGPUs for the time being...
Optimize Expedited Grace Periods

- **Partition**
  - Use the rcu_node combining tree!

- **Batch**
  - Need a mechanism to piggyback off others' expedited grace periods

- **Weaken**
  - My normal advice would be to use RCU, but this *is* RCU...

- **Hardware**
  - Need to be portable, so no FPGAs or GPGPUs for the time being...

- **We therefore stick with partitioning and batching**
Partitioning Expedited Grace Periods
Partitioning Expedited Grace Periods

- `struct rcu_state`
- `struct rcu_node`
- `struct rcu_data` (CPU 0)
- `struct rcu_data` (CPU 15)
- `struct rcu_data` (CPU 4080)
- `struct rcu_data` (CPU 4095)

- `->exp_funnel_mutex`

Diagram showing the relationships between these structures, indicating how expedited grace periods are partitioned.
Partitioning Expedited Grace Periods

But we still have lock-contention bottleneck at root rcu_node structure!!!
Batching Expedited Grace Periods
Batching Expedited Grace Periods
Batching Expedited Grace Periods: Numbering

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

- Expedited Grace Period A
- Expedited Grace Period B
- Expedited Grace Period C
- Expedited Grace Period D

Time
Batching Expedited Grace Periods: Using Numbering

- Start at zero, wait until two
- Start at one, wait until four
- Start at two, wait until four
- Start at three, wait until six
- Start at four, wait until six
- Start at five, wait until eight
- Start at six, wait until eight

General rule: wait = (start + 3) & ~0x1
Batching Expedited Grace Periods: Using Numbering

- Snapshot expedited grace-period sequence number (egpsn)
  - Add three and clear low-order bit

- Acquire locks to start grace period
  - If egpsn has reached snapshot, done!
  - Release locks and exit

- Increment egpsn

- Start expedited grace period

- Wait for expedited grace period to complete

- Increment egpsn
Batching Expedited Grace Periods: Using Numbering

struct rcu_state

struct rcu_node

struct rcu_data
CPU 15

struct rcu_data
CPU 0

struct rcu_data
CPU 4095

struct rcu_data
CPU 4080

Acquire lock, release prior lock, check egpsn

Acquire lock, release prior lock, check egpsn

Acquire lock, check egpsn

Snapshot egpsn
Batching Expedited Grace Periods: Using Numbering

```
struct rcu_state

struct rcu_node

struct rcu_node

struct rcu_node

struct rcu_data CPU 0

struct rcu_data CPU 15

struct rcu_data CPU 4080

struct rcu_data CPU 4095
```

Optimization: Try acquiring root-level lock first, fall back if unavailable.
Expedited Grace Period Example
Expedited Grace Period Example

EGPSN: 0

Not Done
Done but unknown
Knows it is done

Lock rcu_node
Lock rcu_node
Lock rcu_node
Snapshot

A: 2  B: 2  C: 2  D: 2
Expedited Grace Period Example

EGPSN: 0

Lock rcu_node

Lock rcu_node

Lock rcu_data

Snapshot
Expedited Grace Period Example

EGPSN: 0

Lock rcu_node

Lock rcu_node

Lock rcu_data

Snapshot
Expedited Grace Period Example

EGPSN: 1

```
A: 2
   B: 2
     E: 2
     I: 2
   F: 2
     K: 4
C: 2
   D: 2
     G: 2
     J: 2
     H: 2

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
```
Expedited Grace Period Example

EGPSN: 2

A: 2

B: 2

C: 2

D: 2

E: 2

F: 2

G: 2

H: 2

K: 4

One expedited grace period serves ten requests!!!
Expedited Grace Period Example

**EGPSN:** 2

- **C:** 2
  - **B:** 2
    - **E:** 2
    - **I:** 2
  - **D:** 2
    - **F:** 2
    - **K:** 4
    - **J:** 2
    - **H:** 2
      - **L:** 4

- **Lock rcu_node**
- **Lock rcu_node**
- **Lock rcu_data**
- **Snapshot**
Expedited Grace Period Example

EGPSN: 2

- E: 2
  - I: 2
    - M: 4
  - F: 2
  - J: 2
    - N: 4
- G: 2
  - L: 4
    - O: 4

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Expedited Grace Period Example

EGPSN: 2

F: 2

M: 4

P: 4

K: 4

Q: 4

L: 4

N: 4

R: 4

O: 4

S: 4

Lock rcu_node

Lock rcu_node

Lock rcu_data

Snapshot

Fully parallel recognition of batching!
Expedited Grace Period Example

EGPSN: 3

Lock rcu_node

Lock rcu_node

Lock rcu_data

Snapshot
Expedited Grace Period Example

This time, one expedited grace period serves nine requests.
Great Performance and Scalability!!!
Great Performance and Scalability!!!
In Theory, Anyway...
Let's Do Some Benchmarking!!!
Let's Do Some Benchmarking!!!
How Hard Can It Be???
Let's Do Some Benchmarking!!!
How Hard Can It Be???

- Tight loops doing `synchronize_sched_expedited()` with other tight loops doing `rcu_read_lock(): rcu_read_unlock()`
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Tight loops doing `synchronize_sched Expedited()` with other tight loops doing `rcu_read_lock()`: `rcu_read_unlock()`
  - Which resulted in horrid grace-period latencies: hundreds of ms!!

- Small update:
  - `rcu_read_lock()`: `cond_resched_rcu_qs(); rcu_read_unlock()`
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Tight loops doing synchronize_sched Expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!!

- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()
  - Some improvement, but still not good

- Make expedited grace periods note interrupt from idle
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!!

- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()
  - Some improvement, but still not good

- Make expedited grace periods note interrupt from idle
  - Still painful

- Pin the looping kthreads to their own CPUs
Let's Do Some Benchmarking!!!

How Hard It Can Be...

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!

- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()
  - Some improvement, but still not good

- Make expedited grace periods note interrupt from idle
  - Still painful

- Pin the looping kthreads to their own CPUs
  - Better, but still not great – and essentially no batching!!!
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Set kthreads doing grace periods to real-time priority
  - Tens of ms instead of hundreds of ms, better, but...

- Get the readers out of the way
  - Not much difference...

- Make cond_resched_rcu_qs() respond to expedited grace period requests
  - Not much difference

- Get IRC from Sasha Levin saying that KASAN complains about address-out-of-range errors
  - What exactly does C do with double subscripts? The wrong thing...
  - So ditch the double subscripts in favor of explicit pointer traversals
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Collect data via ftrace rather than printk
  - Gets rid of some preemptions...
  - Still greater than 10 milliseconds worst case, so look at ftrace!

- New arrivals jumping the queue!!!
Queue-Jumping Problem

EGPSN: 4

L: 4

K: 4

M: 4

Q: 4

N: 4

S: 4

P: 4

T: 6

R: 4

U: 6

Lock rcu_node

Lock rcu_node

Lock rcu_data

Snapshot
Queue-Jumping Problem

EGPSN: 4

V: 6

K: 4

M: 4

P: 4

Q: 4

T: 6

N: 4

R: 4

O: 4

S: 4

U: 6

Lock rcu_node

Lock rcu_node

Lock rcu_data

Snapshot
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Collect data via ftrace rather than printk
  - Gets rid of some preemptions...
  - Still greater than 10 milliseconds worst case, so look at ftrace!

- New arrivals jumping the queue!!!
  - So eliminate the queue-jumping optimization
  - But only minor improvements in worst case and in batching

- New arrivals still jumping the queue due to wakeup latency
Queue-Jumping Problem Redux

EGPSN: 5

K: 4
M: 4
P: 4

O: 4
Q: 4
T: 6

N: 4
R: 4
U: 6

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Queue-Jumping Problem Redux

EGPSN: 6

\[ \text{V: 6} \]

\[ \text{K: 4} \]
\[ \text{O: 4} \]

\[ \text{M: 4} \]
\[ \text{Q: 4} \]
\[ \text{N: 4} \]
\[ \text{S: 4} \]

\[ \text{P: 4} \]
\[ \text{T: 6} \]
\[ \text{R: 4} \]
\[ \text{U: 6} \]

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Queue-Jumping Problem Redux

EGPSN: 6

K: 4
O: 4
S: 4
U: 6

M: 4
Q: 4
N: 4
P: 4
T: 6
R: 4

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Queue-Jumping Problem Redux

EGPSN: 6

M: 4  Q: 4  N: 4

P: 4  T: 6  R: 4

Wakeup delay can be significant, and in the meantime...
Queue-Jumping Problem Redux

EGPSN: 6

M: 4  Q: 4  N: 4  V: 8
P: 4  T: 6  R: 4

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Queue-Jumping Problem Redux

EGPSN: 6

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Queue-Jumping Problem Redux

EGPSN: 6

K: 4
M: 4
P: 4
Q: 4
T: 6

V: 8
N: 4

Lock rcu_node
Lock rcu_node
Lock rcu_data
Snapshot
Queue-Jumping Problem Redux

EGPSN: 6

Tasks K, M, P, Q, and T stuck waiting on Task V!!!
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Collect data via ftrace rather than printk
  - Gets rid of some preemptions...
  - Still greater than 10 milliseconds worst case, so look at ftrace!

- New arrivals jumping the queue!!!
  - So eliminate the queue-jumping optimization
  - But only minor improvements in worst case and in batching

- New arrivals still jumping the queue due to wakeup latency
  - So switch from mutex to rt_mutex (worry about mainlining later...)
  - Much better!!! 6x batching on four CPUs, sub-10-ms latencies
  - But 4.7 milliseconds is not exactly expedited...
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Automation causes entire benchmark to run at boot time
  - Not the best time for low OS jitter!
  - Delay the test until after boot completes (after a few false starts)
  - Maximum grace-period latency below 1ms, good batching
  - But getting RCU CPU stall warnings and RT throttling

- So put thread to SCHED_OTHER before ftrace_dump(), get rid of readers, and delay before ftrace_dump()
  - 99th percentile at 10 microseconds, max at about 500 microseconds
  - More like it!
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Automation causes entire benchmark to run at boot time
  - Not the best time for low OS jitter!
  - Delay the test until after boot completes (after a few false starts)
  - Maximum grace-period latency below 1ms, good batching
  - But getting RCU CPU stall warnings and RT throttling

- So put thread to SCHED_OTHER before ftrace_dump(), get rid of readers, and delay before ftrace_dump()
  - 99th percentile at 10 microseconds, max at about 500 microseconds
  - More like it!

- But six CPUs is a small fraction of 4096 CPUs!!!
Benchmarking on 4096 CPUs
Benchmarking on 4096 CPUs

- I don't actually have access to a 4096-CPU system
  - Just to the bug reports filed by people who do have such systems
- But, as noted in the past, I have relevant experience:
Benchmarking on 4096 CPUs

- I don't actually have access to a 4096-CPU system
  – Just to the bug reports filed by people who do have such systems
- But, as noted in the past, I have relevant experience:
Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu Expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- But extremely long runtimes for 256 tasks on 32 CPUs...
Benchmarking on 4096 CPUs

- Dirty trick #1: Note that `synchronize_rcu_expedited()` blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs

- But extremely long runtimes for 256 tasks on 32 CPUs...
  - Problem: Tasks with enough measurements compete for CPU time with those that are not yet done
    - But we need them to be running in order to provide needed load
    - Just not at realtime priority
  - Solution: Once a given task has enough measurements, drop it to non-realtime priority
    - Allows scheduler to determine which tasks are important
    - Decreases runtime by more than a factor of three
    - So that I might be able to collect enough data in time for this talk!!!
## Dirty Trick #1 Results (32 CPUs, 256 Tasks)

<table>
<thead>
<tr>
<th>Min</th>
<th>Mean</th>
<th>99&lt;sup&gt;th&lt;/sup&gt; Percentile</th>
<th>Maximum</th>
<th>Batching</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 us</td>
<td>35.6 us</td>
<td>276 us</td>
<td>806 us</td>
<td>68.9</td>
</tr>
<tr>
<td>3 us</td>
<td>40.7 us</td>
<td>284 us</td>
<td>512 us</td>
<td>81.6</td>
</tr>
<tr>
<td>1 us</td>
<td>59.3 us</td>
<td>257 us</td>
<td>1146 us</td>
<td>149.6</td>
</tr>
</tbody>
</table>
Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu Expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs

- Dirty trick #2: Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
Dirty Trick #2 Results (54 CPUs, 256 Tasks, 4 Levels)

<table>
<thead>
<tr>
<th>Min</th>
<th>Mean</th>
<th>99&lt;sup&gt;th&lt;/sup&gt; Percentile</th>
<th>Maximum</th>
<th>Batching</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 us</td>
<td>591.5 us</td>
<td>3492 us</td>
<td>5562 us</td>
<td>96.8</td>
</tr>
<tr>
<td>9 us</td>
<td>597.8 us</td>
<td>3777 us</td>
<td>5859 us</td>
<td>97.8</td>
</tr>
<tr>
<td>108 us</td>
<td>6739.5 us</td>
<td>34021 us</td>
<td>38133 us</td>
<td>126.2</td>
</tr>
<tr>
<td>59 us</td>
<td>12610.5 us</td>
<td>86876 us</td>
<td>140910 us</td>
<td>130.7</td>
</tr>
<tr>
<td>11 us</td>
<td>797.8 us</td>
<td>5127 us</td>
<td>11827 us</td>
<td>105.0</td>
</tr>
<tr>
<td>6 us</td>
<td>568.0 us</td>
<td>2254 us</td>
<td>5042 us</td>
<td>80.1</td>
</tr>
</tbody>
</table>

Horrible results, probably due to new interactions in the taller tree. And greater interference from other users on this shared machine.
## Dirty Trick #2 Results (54 CPUs, 256 Tasks, 2 Levels)

<table>
<thead>
<tr>
<th>Min</th>
<th>Mean</th>
<th>99th Percentile</th>
<th>Maximum</th>
<th>Batching</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 us</td>
<td>220.2 us</td>
<td>553 us</td>
<td>690 us</td>
<td>182.2</td>
</tr>
<tr>
<td>6 us</td>
<td>169.4 us</td>
<td>1034 us</td>
<td>1558 us</td>
<td>178.1</td>
</tr>
<tr>
<td>5 us</td>
<td>166.9 us</td>
<td>1177 us</td>
<td>3025 us</td>
<td>111.7</td>
</tr>
</tbody>
</table>

Increased confidence of likely new interactions in the taller tree. And greater interference from other users on this shared machine.
Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu Expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs

- Dirty trick #2: Decrease fanouts to obtain a full-height RCU
  combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
  - But lab machine uses rotating rust, and it therefore takes a good long
    time to dump out the ftrace data
  - Longer-term fix: Do the data reduction in the kernel
Benchmarking on 4096 CPUs

- **Dirty trick #1:** Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs

- **Dirty trick #2:** Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
  - But lab machine uses rotating rust, and it therefore takes a good long time to dump out the ftrace data
  - Longer-term fix: Do the data reduction in the kernel
  - Even longer-term fix: Use a system with 4096 real CPUs
Benchmarking on 4096 CPUs

- **Dirty trick #1**: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs

- **Dirty trick #2**: Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
  - But lab machine uses rotating rust, and it therefore takes a good long time to dump out the ftrace data
  - Longer-term fix: Do the data reduction in the kernel
  - Even longer-term fix: Use a system with 4096 real CPUs

- More dirty tricks will likely be required!
Summary and Lessons (Re)learned
Summary and Lessons (Re)learned

- Benchmarking is not as easy as it looks ;-)  
  - Obvious optimizations often aren't  
    - Uncontended-case fastpath to root node problematic  
- Maintaining request order is important in this case  
  - Which is unfortunate, as this can be complex and expensive  
- Fixed a couple of performance bugs:  
  - Make expedited grace period IPI handlers check for idle  
  - Make cond_resched_rcu_qs() satisfy expedited grace periods  
  - And I have at least one more to fix!  
- At the end of the day, real full-scale testing is needed  
  - There are likely to be other performance bugs  
    - IPIs sent serially, wakeups likely to be a bottleneck, ...  
  - But it is good to get a couple of them out of the way!!!
Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
Questions?