QPW: How to improve latency and CPU Isolation without cost

Leonardo Brás Soares Passos

Linux Plumbers Conference 2024

whoami

- Leonardo Brás Soares Passos
- Work @ Red Hat (Virt-team)
  - Linux Kernel
    - Improving CPU Isolation & RT
    - Reducing RT guest latency
    - Improving RISC-V arch code (as a side quest)
- Find me: leobras @ {redhat.com, GitLab, GitHub, IRC}

Introduction

- What we want?
  - To run time-sensitive tasks with very low latency
- How can we achieve this?
  - Running the RT task as uninterrupted as possible.
- If it’s running, it will be more responsive
- If not, there are latency costs of switching contexts

One problem for RT

- Interruptions unrelated to RT task:
  - Increases latency on that CPU
- schedule_work_on(cpu) & queue_work_on(cpu)
  - Causes Inter Processor Interruptions (IPIs) on target cpu
- We can use only housekeeping CPUs for some of them
- Can we somehow avoid the rest?

Use of per-cpu caches

- This is a very efficient strategy for sharing global resources on SMP systems:
  - Each CPU using the resource gets a per-cpu cache
  - Allocation and freeing resources happen in the local cache
  - When local cache is full (or empty), it accesses the global cache for expanding (or shrinking) the local cache.
    - This reduces the occurrences of global locking & contention
  - Used in memcg, slub, swap.
- Issue: Actively reclaiming resources from remote per-cpu caches requires schedule_work_on(all_online_cpus).
  - An IPI for each online cpu is issued, interrupting the work there

The generic code

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

The generic code

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);


/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

Generates an IPI for a remote CPU

The generic code

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

Bad for latency on that CPU

Generates an IPI for a remote CPU

Getting rid of the schedule_work_on()

- Replace local_locks() with per-cpu spinlocks()
  - Get local CPU's spinlock() for each local operation
  - Get remote CPU's spinlock() for remote operation
    - Instead of schedule_work_on() that cpu
- Remote operations don't happen very often
  - Contention on per-cpu spinlocks() should be very rare.
- Some work done on this, by Mel Gorman[1]:
  - 01b44456a7aa7 ("mm/page_alloc: replace local_lock with normal spinlock")

local_lock + IPI → spinlock

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

```
/* Hotpath: work locally */
spin_lock(s->lock);
do_local_work_on(s);
spin_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        p = per_cpu_ptr(mystruct, cpu);
        spin_lock(p->lock)
        p->work(p);
        spin_unlock(p->lock)
}
```

Wait, are not spinlock() expensive?

- Most of a spinlock()’s cost comes from:
  - Contention
  - Getting cacheline exclusiveness
  - Memory barriers

Wait, are not spinlock() expensive?

- Most of a spinlock()’s cost comes from:
  - Contention
    - Not expected, since remote operations don’t happen often
  - Getting cacheline exclusiveness
  - Memory barriers

Wait, are not spinlock() expensive?

- Most of a spinlock()’s cost comes from:
  - Contention
    - Not expected, since remote operations don’t happen often
  - Getting cacheline exclusiveness
    - Local CPU will mostly have that per-cpu spinlock()’s cacheline exclusiveness already, since remote operations don’t happen often
    - Invalidation will only happen after a remote operation
  - Memory barriers

Wait, are not spinlock() expensive?

- Most of a spinlock()'s cost comes from:
  - Contention
    - Not expected, since remote operations don't happen often
  - Getting cacheline exclusiveness
    - Local CPU will mostly have that per-cpu spinlock()'s cacheline exclusiveness already, since remote operations don't happen often
    - Invalidation will only happen after a remote operation
  - Memory barriers
    - Are not supposed to be that expensive

Wait, are not spinlock() expensive?

- From previous studies [2]:
  - Switching from local_locks to per-cpu spinlocks comes with a cost of 3-15 extra cycles per lock/unlock (x86_64, ARM64)
    - local access only, after a remote access it will cost a cache bounce
  - That may be too much for hotpaths
- But on PREEMPT_RT=y
  - local_locks are already per-cpu spinlocks.
  - Above costs are already paid for, so why not?
    - Grab remote spinlock, do the work, release

QPW: Queue PerCPU Work (on)

QPW

- Create an interface that allow:
  - Keep working the same way on PREEMPT_RT=n
  - Apply the new strategy in PREEMPT_RT=y
    - For cases in which it applies

- It requires:
  - A new helper to get the requested work done
  - A new way of getting the remote cpu’s “local_lock”

QPW: Implementation [3]

- qpw locks:
  - Replace local_locks only on functions that can be remotely called
    - PREEMPT_RT=n → local_locks
    - PREEMPT_RT=y → per-cpu spinlocks (of the remote cpu)
  - Uses the per-cpu spinlock already available in local_lock_t

- queue_percpu_work() & flush_percpu_work()
  - Replace non-percpu functions
    - On uses we are sure not to touch the local hardware resources
    - PREEMPT_RT=n → use the non-percpu functions
    - PREEMPT_RT=y → grab target cpu spinlock, do the work, unlock

local_lock + IPI → spinlock

```
struct qpw_struct {
    struct work_struct work;
    int cpu;
};

qpw_lock(lock, cpu){
    spin_lock(per_cpu_ptr(lock, cpu));
}

qpw_unlock(lock, cpu) {
    spin_unlock(per_cpu_ptr(lock, cpu));
}
```

```
queue_percpu_work_on(cpu, qpw) {
    p = qpw->work;
    p->func(qpw);
}

flush_percpu_work_on(qpw) {
    /* do nothing */
}
```

Bugs that would vanish

```
[342431.665417] INFO: task grub2-probe:24484 blocked for more than 622 seconds.
[342431.665515] task:grub2-probe state:D  stack:0  pid:24484 ppid:24455 flags:0x00004002
[342431.665523] Call Trace:
[342431.665525]  <TASK>
[342431.665527]  __schedule+0x22a/0x580
[342431.665537]  schedule+0x30/0x80
[342431.665539]  schedule_timeout+0x153/0x190
[342431.665543]  ? preempt_schedule_thunk+0x16/0x30
[342431.665548]  ? preempt_count_add+0x70/0xa0
[342431.665554]  __wait_for_common+0x8b/0x1c0
[342431.665557]  ? __pfx_schedule_timeout+0x10/0x10
[342431.665560]  __flush_work.isra.0+0x15b/0x220
```

QPW: Implementation [3]

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);


/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
flush_work(s->work);
```

```
/* Hotpath: work locally */
qpw_lock(s->lock, n);
do_local_work_on(s);
qpw_unlock(s->lock, n);


/* Eventually do remote work */
for_each_online_cpu(cpu){
        queue_percpu_work_on(cpu, s->qpw);
}
flush_percpu_work(s→qpw);
```

QPW: Implementation [3]

- Selected 3 examples for testing in original patchset
  - memcontrol, slub, swap
  - Tested on 128-cpu x86_64 machine, 24-cpu ARM64 machine
  - Works fine, reduces latency.

- Test results: (ARM64 machine)
  - Cyclictest: Max latency 58us → 40us
  - Oslat: Max latency 3us → 2us

QPW: Implementation [3]

- Advantages:
  - Can convert usages on demand
  - Does not mess with other uses

- Disadvantages:
  - Need to convert case by case, desired function needs to receive a cpu parameter
  - Need to be sure the remote work won’t touch local cpu data
    - Can be hard to guarantee

QPW: Another implementation

- “Emulate” this_cpu & smp_processor_id
  - Create a new field ecpu in thread_info
    - Use it to get the results of this_cpu* and smp_processor_id
  - Change it on:
    - Entry of queue_percpu_work_on() to the “emulated” cpu number
    - Exit of queue_percpu_work_on() to the physical cpu number
    - On task migration, change the ecpu to new physical cpu
      - Only if ecpu was the same as the old physical cpu number
- To convert a case, just rename
  - queue_work_on() → queue_percpu_work_on()
  - flush_work() → flush_percpu_work()

QPW: Another implementation

- Advantages:
  - More elegant approach
    - No new structs or locks, just a couple helpers
      - queue_percpu_work_on() and flush_percpu_work(), which is empty.
  - Much easier to convert functions
  - Can replace regular workqueue functions
    - If we create a raw version that uses actual cpu number & apply it on cases that deal with local hardware resources.

- Disadvantages:
  - A little more overhead than the previous,
    - Requires a per-cpu spinlock to avoid nested emulation (emulate + preempt + emulate)
  - Require either to create an emulation layer, or to unify the way archs implement this_cpu() and smp_processor_id()
    - All but x86, s390 & ppc64 already use a cpu field in thread_info

QPW: Another points

- Enable / Disable by compile-time option
  - And / or boot parameter
- Make it work only on Isolated CPUs
  - Need to test latency & performance

- <Your suggestion here>

Thanks!

Questions?
Suggestions?

References:

[1] https://lore.kernel.org/all/20220624125423.6126-8-mgorman@techsingularity.net/
[2] https://lpc.events/event/17/contributions/1484/
[3] https://lore.kernel.org/all/20240622035815.569665-1-leobras@redhat.com/

Next topic

Improving guest latency & throughput by improving RCU in KVM

Leonardo Brás Soares Passos
Linux Plumbers Conference 2024

Introduction

- What we want?
  - To run time-sensitive tasks with very low latency
    - Inside KVM guests!

- What's the issue?
  - Getting latency violations in cyclictest & oslat
  - Cause: rcuc/N thread running in same CPU as the RT task
    - What's rcuc/N?

Example

```
trace.dat:    CPU 2/KVM-1472961 [004] d..2. 85031.708830: sched_switch:    CPU 2/KVM:1472961 [98] R ==> rcuc/4:62 [95]
trace.dat:      rcuc/4-62       [004] d..2. 85031.708844: sched_switch:    rcuc/4:62 [95] S ==> CPU 2/KVM:1472961 [98]
trace.dat:    CPU 2/KVM-1472961 [004] d..1. 85031.708854: kvm_entry:       vcpu 2 rip 0xffffffffa8c4eac2
trace.dat:    CPU 2/KVM-1472961 [004] d..1. 85031.708857: kvm_exit:        reason PREEMPTION_TIMER rip 0xffffffffa8... info 0 0
trace.dat:    CPU 2/KVM-1472961 [004] d..1. 85031.708859: kvm_entry:       vcpu 2 rip 0xffffffffa8c4eac2
trace.dat:    CPU 2/KVM-1472961 [004] d..1. 85031.708869: kvm_exit:        reason MSR_WRITE rip 0xffffffffa806f3a4 info 0 0
trace.dat:    CPU 2/KVM-1472961 [004] d..1. 85031.708870: kvm_entry:       vcpu 2 rip 0xffffffffa806f3a6
trace-3.dat:      <idle>-0      [002] d.h1. 85031.708874: local_timer_entry:   vector=236
trace-3.dat:      <idle>-0      [002] d.h1. 85031.708876: hrtimer_expire_entry: hrtimer=0xffff... now=54... function=hrtimer_wakeup/0x0
trace-3.dat:      <idle>-0      [002] dNh1. 85031.708877: hrtimer_expire_exit:  hrtimer=0xffff9d910079be18
trace-3.dat:      <idle>-0      [002] dNh1. 85031.708882: local_timer_exit:    vector=236
trace-3.dat:      <idle>-0      [002] d..2. 85031.708884: sched_switch:      swapper/2:0 [120] R ==> cyclictest:1599 [4]
trace-3.dat:   cyclictest-1599  [002] ..... 85031.708905: print:             tracing_mark_write: hit latency threshold (63 > 50)
```

rcuc/N thread

- Threads that run rcu_core when needed
  - There is one per CPU, and it’s invoked by timer interrupt
  - They help RCU to work when the system is busy

- Needs to run if a quiescent state took to long to happen on that CPU, after a grace period

rcuc/N thread

- Threads that run rcu_core when needed
  - There is one per CPU, and it’s invoked by timer interrupt
  - They help RCU to work when the system is busy

- Needs to run if a quiescent state took to long to happen on that CPU, after a grace period

- *RCU ? Quiescent state? Grace period? What?*

RCU [1]

- It’s a very efficient parallel programming mechanism
  - On read: Very efficient, requires no atomic operations
  - On write: replace protected memory with a new one
    - Then it waits until no other CPU is reading the old memory
    - Before it can free it and/or continue the procedure
- When not deferring any RCU-protected memory:
  - CPU is said to be in quiescent state

The issue

- If a CPU stays too long without reporting a quiescent state, the running process needs to be interrupted so that CPU can report, and the waiting CPU can get unstuck.

- That long running task is exactly the case of a guest vCPU, which is running an RT task on an isolated CPU, and pooling for network, for example.

The issue

- After the guest is running for some time, and a quiescent state is required on that CPU:
  - Timer interrupt provokes guest_exit()
  - Timer handler checks RCU needs
    - And then sched-in that cpu’s rcuc/N thread
  - After it finishes reporting the quiescent state, it scheds-in the guest vCPU again

- All this procedure causes a lot of latency into the task

The solution

- Guest running state is considered an extended quiescent state, as RCU-protected areas are not used for a long time.
  - KVM reports a quiescent state on guest entry, but for some reason, not on guest exit.
- So, report a quiescent state in guest exit, so every pending quiescent state reporting request that happened while the guest ran gets satisfied, and rcuc/N doesn’t need to run. [2]

The solution

- This solution reduces a lot the reproduction rate, but it still happens sometimes.
  - Reason: Any CPU can request a quiescent state report between guest_exit and the timer interrupt handler checking, and it this will cause rcuc/N to wake, since there is a new quiescent state request.
- The solution on top of the solution is RCU patience[3]:
  - A new command-line option that allows the kernel to wait for a certain time since the oldest valid unreported quiescent state request before waking up rcuc/N.

Results [4]

- Latency improvement:
  - Max latency on guest cyclictest went from 58us → 37us
- Performance gains in RT host
  - There were marginal gains in cpu cycles inside the VM (~0.6%), due to number of guest_exit and time spent inside the guest balancing themselves
- Performance gains in non-RT host
  - Both the average time spend inside the VM and the number of VM entries raised, causing the VM to have **~4.5%** more cpu cycles available to run it’s workload

Conclusion

- On top of latency improvement, this change could also achieve almost 5% improvement in cpu time available for VMs at non-RT kernels, so it may be of interest to those who sell VM time.

- This solution was merged in mainline as follows:
  - RCU/KVM [2] → Merged on 2024-09-06
  - RCU Patience [3] → Merged on 2024-07-15

Thanks!

Questions?
Suggestions?

References:

[1] https://www.kernel.org/doc/Documentation/RCU/rcu.txt
[2] Commit 593377036e50 ("kvm: Note an RCU quiescent state on guest exit")
[3] Commit 68d124b09999 ("rcu: Add rcutree.nohz_full_patience_delay to reduce nohz_full OS jitter")
[4] https://lore.kernel.org/all/ZnPUTGSdF7t0DCwR@LeoBras/