ORACLE

Dynamic paravirt lock-ops

For a dynamic world

Ankur Arora
ankur.a.arora@oracle.com

Contents

Guests should be more dynamic

```
KVM_HINTS_REALTIME=1 => native pv_lock_ops[1]
KVM_HINTS_REALTIME=0 => paravirt pv_lock_ops[2]
```

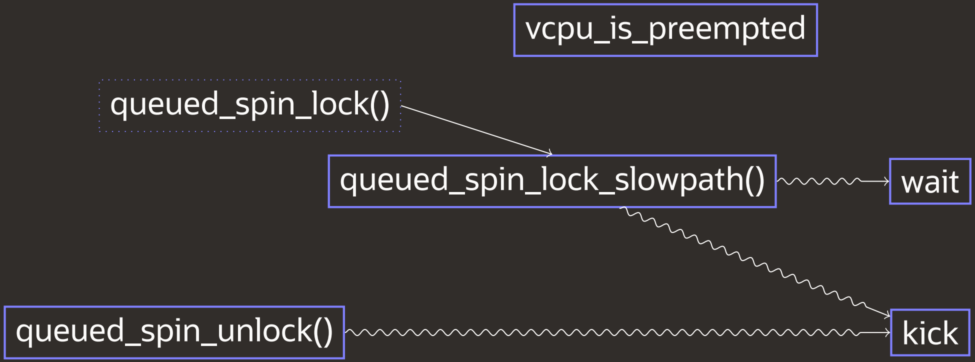
A guest that starts with `KVM_HINTS_REALTIME`, and then becomes oversubscribed: typically ends in softlockups.

The recommended fix is that a host should only advertise `KVM_HINTS_REALTIME` if it can guarantee it for the lifetime of the guest.

[1]Hypervisor specific ops are `paravirt_nop()`
[2]KVM's hypervisor specific ops are `kvm_wait()`, `kvm_vcpu_kick()`

`pv_lock_ops` state-machine

Queued spinlocks are based on MCS locks



Why don’t we just use paravirt locks all the time?

Unlock fastpath

- `movb $0, (%rdi)`
- `lock cmpxchg %dl,(%rdi)`

Paravirt `queued_spin_lock_slowpath` are pessimistic by default

What does switching `pv_lock_ops` involve?

Switch all call-sites, kernel and modules, for all 5 interfaces atomically[3]

Transform back and forth between instruction sequences like this one for `queued_spin_unlock`

```
__native_queued_spin_unlock:
 0: c6 07 00                movb   $0x0,(%rdi)
 3: 0f 1f 40 00             nopl   0x0(%rax)

__pv_queued_spin_unlock:
 0: e8 31 e6 ff ff          callq  0xffffffffffffe636
 5: 66 90                   xchg   %ax,%ax
```

[3] `queued_spin_unlock()` is almost always inlined

What does switching `pv_lock_ops` involve? (contd.)

- Other ops might go from `CALL+NOP2 (pv_lock_ops.wait = kvm_wait())` to `NOP7 (pv_lock_ops.wait = paravirt_nop())` or back
- Spinlocks cannot sleep:
  - So no references to stale `pv_lock_ops` opcodes on the stack

Active users

… while we are patching

Contexts

- Tasks
- Softirqs
- Interrupt handlers
  - IPIs in particular get used from `text_poke_bp()` while patching
- NMI handlers

All of the above, but nested.

Remember, I said spinlocks (thus `pv_lock_ops`) cannot *sleep*.

- They can get context switched out in the hypervisor

Mechanism: INT3

Patching while potentially executing code that you are patching. Enter `INT3`.

- Standard Linux mechanism for modifying cross-modifying code
- Single byte breakpoint instruction: opcode `0xCC`
- Used as a barrier at entry
  - Assuming a single entry point to instruction sequence
- If this barrier is hit, the control flow shifts to an INT3 handler
  - Which emulates the original target or the new

V1: approach

Use `stop_machine()`

- lock-step state-machine on all the VCPUs
  - Inner loop on CPU-patcher and all the secondary CPUs waiting to synchronize at each step.
  - interrupts are disabled
  - no IPIs needed for `sync_core()`
  - no `pv_lock_ops` on the stack: all VCPUs are executing in `stop_machine()`
- NMIs are the only risk
  - the INT3 handler also implements a subset of this state-machine, so we can make forward progress if the primary or a secondary CPU hit an NMI[4]

[4]Multiple simultaneous NMIs complicate the handling somewhat.

V1: state-machine

```
CPU-patcher          CPU-x
                             /* CALL, NOP2:  e8 31 e6 ff ff 66 90 */
write-INT3                   /* INT3, ... :  cc 31 e6 ff ff 66 90 */
sync()               smp-cond-load-acquire(state == INT3-written)
                     sync()
write-rest                   /* INT3, ... :  cc 07 00 0f 1f 40 00 */
sync()               smp-cond-load-acquire(state == rest-written)
                     sync()
write-first-byte             /* MOV, NOP4 :  c6 07 00 0f 1f 40 00 */
sync()               smp-cond-load-acquire(state == first-written)
                     sync()
```

V1: last words

It worked, but ... `stop_machine()`

A review comment said: ”bonghits crazy code.”

Which was pretty understated, in hindsight.

V2: design

Where I magically discover a less crazy way to do this...

Step1: prefix INT3

- Use `INT3 (0xCC)` as a site-local barrier; everywhere
- Allows us to enforce atomicity while patching multiple sites

Step2: global ”barrier”

- Divides the guest state into pre and post stages: old and new `pv_lock_ops`
- Meanwhile the INT3 handler emulates old or new `pv_lock_ops`

Step3: finish patching

- Use the protection offered by `INT3 (0xCC)` to finish writing the new opcodes

V2 design: global barrier

Sure, but how...

The transition point in old to new `pv_locks_ops` requires a point where no ops are executing.

A CPU after crossing the barrier:

- Counts all spinlocks[5] under execution
- Counting happens in the INT3 handler

[5] Figuratively, not literally. We have no way of counting locks, we can, however, count entry in `queued_spin_slowpath()` and exit in `queued_spin_unlock()`

V2 design: global barrier

```
atomic_t barrier_cpus, active_lock_ops;
DEFINE_PER_CPU(int, paravirt_switch_barrier);
void patch_barrier(void) {
     this_cpu_write(paravirt_switch_barrier, 1);
     atomic_inc(&barrier_cpus);
      /* Count active_lock_ops if this_cpu_read(paravirt_switch_barrier). */
}
```

Property that needs to hold:

```
    atomic_read(&barrier_cpus) == num_online_cpus
      && atomic_read(&active_lock_ops) == 0;
```

Once this holds, INT3 handling can start emulating the new `pv_lock_ops` and move to Step 3.

V2 design: more on counting

What are we counting?

- *not* counting the fastpath: `queued_spin_lock()` … `queued_spin_unlock()`
- count the slowpath: `queued_spin_lock_slowpath()` … `queued_spin_unlock()`

Use a bitmap to be able to tell the two calls to `queued_spin_unlock()` apart.

Note that spinlocks can be arbitrarily nested, in each of the four contexts (thread, softirq, interrupt, NMI)

Show me the code

V2:

- https://github.com/terminus/linux/tree/alternatives-v2

V1:

- https://github.com/terminus/linux/tree/alternatives-rfc-upstream-v1
- https://lore.kernel.org/lkml/20200408050323.4237-1-ankur.a.arora@oracle.com/

Questions?

Or send them to ankur.a.arora@oracle.com.