Using KVM as a Real-Time Hypervisor

Jan Kiszka, Siemens AG, CT T DE IT 1
Corporate Competence Center Embedded Linux
jan.kiszka@siemens.com
Agenda

- Motivation & scenarios
- RT benchmark updates
- Improving QEMU RT performance
  - Analysis of critical paths
  - Steps to overcome latency spots
- Summary
Recall Last Year:
Why Using KVM in Embedded?

“We just need a tiny hypervisor to fully exploit this multicore CPU”
- “A few thousand” lines of hypervisor code
- Minimal hardware emulation
- “A bit” paravirtualization
- Devices are passed through

“Why Using KVM in Embedded?”
- over-commit resources
- manage power
- freeze / migrate guests
- use advanced HA features
- ...

```
RTOS   Linux   Windows  $OS
```

```
Hypervisor
Core 1  Core 2  Core 3  Core n
```
Recall Last Year:
Why Using KVM in Embedded?

“We just need a tiny hypervisor to fully exploit this multicore CPU”
- “A few thousand” lines of hypervisor code
- Minimal hardware emulation
- “A bit” paravirtualization
- Devices are passed through

“But it would be nice to...”
- share some devices
- run upstream Linux and latest Windows
Recall Last Year:
Why Using KVM in Embedded?

“We just need a tiny hypervisor to fully exploit this multicore CPU”
- “A few thousand” lines of hypervisor code
- Minimal hardware emulation
- “A bit” paravirtualization
- Devices are passed through

“But it would be nice to...”
- share some devices
- run upstream Linux and latest Windows
- over-commit resources
- manage power
“We just need a tiny hypervisor to fully exploit this multicore CPU”
- “A few thousand” lines of hypervisor code
- Minimal hardware emulation
- “A bit” paravirtualization
- Devices are passed through

“But it would be nice to...”
- share some devices
- run upstream Linux and latest Windows
- over-commit resources
- manage power
- backup / migrate guests
- use advanced HA features
- ...
...and in Real-Time Scenarios?
Pros & Cons

From partitioning hypervisors...
✚ High degree of temporal isolation
✚ Static allocations simplify RT guarantees
− Poor flexibility
− Non-commodity setup
...and in Real-Time Scenarios?
Pros & Cons

From partitioning hypervisors...
✚ High degree of temporal isolation
✚ Static allocations simplify RT guarantees
− Poor flexibility
− Non-commodity setup

... to full virtualization
− Usually not designed for RT
− Higher complexity makes establishing RT harder
✚ Benefit from large user base
  ▪ Guest support
  ▪ Test coverage
✚ Benefit from advanced virtualization features
✚ RT and SMP scalability share many requirements
Typical Real-Time Guest Setups

Guest types
- Classic RTOS
- AMP (RTOS + x)
- GPOS with RT requirements

Guest interacts with real world – in real-time
- Real-time network (normal/RT Ethernet, fieldbuses, etc.)
- Digital & analogue I/O interfaces
- Data acquisition adapters

Interface access
- Pass-through, i.e. 1:1 mapping of periphery to guest
- Emulation
  - Decoupling of guest driver and host hardware
  - Physical interface sharing – or avoiding (test environments)
Benchmark Updates

What is possible today?
Timed Task Benchmarks: Setup (1)

Host system
- Intel Core i7, 4 cores, 2 threads each
- OpenSUSE 11.4
- PREEMPT-RT kernel 2.6.33.9-rt31
- `cyclictest` measures timed task wakeup latency
  `cyclictest -n -p 99 -h 500 -q`
- Host-side load
  - Cache benchmark loop
    `calibrator 3392 8M outputfile`
  - I/O benchmark loop
    `echo 1 > /proc/sys/vm/drop_caches ; bonnie -y -s 2000`
  - Load loops and cyclictest (for host benchmark) or guest VCPU thread (for guest benchmark) bound to host CPU 0
Timed Task Benchmarks: Setup (2)

Guest system
- OpenSUSE 11.4
- PREEMPT-RT kernel 2.6.33.9-rt31
- qemu-kvm patched to allow prioritization
- VM configured to avoid latency-sensitive guest exits:
  -m 1G -drive file=guest.img,if=virtio
  -rt maxprio=80,paioprio=1 -nographic -vga none
  -netdev user,hostfwd=:2222-:22,id=net
  -net nic,netdev=net
- cyclicstest measures timed guest task wakeup latency
  cyclicstest -n -p 99 -h 500 -q
- Host-side load kept unchanged
Timed Task Benchmarks:
Results after ~3h

- Cyclic test on host
  - Maximum: 29 µs
  - Average: 1 µs

- Cyclic test on guest
  - Maximum: 112 µs

Note: Test length too short for reliable maxima
External Event Benchmark: AMP RT Guest with Passed-Through NIC

Host configuration
- Base setup as before
- Intel i82541PI NIC as I/O device (no MSI)
- VM with 2 VCPUs

Guest properties
- GPOS and RTOS on different VCPUs
- RTOS only interacts with
  - APIC & IO-APIC
  - Assigned devices (here: PCI NIC)
  => no exits to QEMU user space
- GPOS requires full-blown virtualization, specifically VGA
External Event Benchmark: Measuring Network Latency

External measurement system
- Linux/Xenomai with RTnet
- rtping @100 HZ

Load scenario
- hackbench 150 process 1000
- Disk I/O load on host
- ping -f from host to GPOS guest (via tap+virtio)
- ftrace enabled for events

Worst case round-trip latency: 330 µs (after 16 h)
External Event Benchmark: Measuring Network Latency

External measurement system
- Linux/Xenomai with RTnet
- rtping @100 HZ

Load scenario
- hackbench 150 process 1000
- Disk I/O load on host
- ping -f from host to GPOS guest (via tap+virtio)
- ftrace enabled for events

Worst case round-trip latency: 330 µs
(after 16 h)

Same scenario with emulated NIC: 100 ms – and more
(prioritized host NIC IRQ & RX Soft IRQ)
QEMU Still Ruining Latencies

Everything under qemu_global_mutex

- Remaining synchronous disk I/O
  
  *Note: observed io_submit() syscall latencies >1 s, paio architecture is immune*

- Network I/O
- Terminal I/O
- X interaction (GUI updates)
- Dirty RAM log synchronization
  
  (>10 ms on synchronize_srcu Expedited)
- ...and probably more

qemu_global_mutex is a no-go for RT code paths!
Overcoming the Global Lock –
Road Works

CPUSTate
- Read/write access
  - cpu_single_env

PIO/MMIO request-to-device dispatching
Coalesced MMIO flushing

Back-end access
- TX on network layer
- Write to character device
- Timer setup, etc.

Back-end events (iothread jobs)
- Network RX, read from chardev, timer signals, …

IRQ delivery
- Raising/lowering from device model to IRQ chip
- Injection into VCPU (if user space IRQ chips)
Step 1: Localize CPUSState

VCPU owns its CPUSState
- No remote write unless VCPU is stopped
- Establish formal rule
  (pre-exists for KVM core)
- Just few code changes required

cpu_current_env becomes per-CPU variable
- pthread_set/get_specific on UNIX
- Win32 requires wrapping
- Works with single TCG CPU thread as well
Step 2: I/O Dispatching

Which device handles accessed memory region?

Critical path
- Walk memory map
- Obtain handler & device reference
- Invoke handler
- Done

Preferred approach: lock-less
- Modifications are rare
- Acquiring read-side lock is costly, may even deadlock

Solution: stop machine while modifying memory map
(pattern also used in kvm-tool)
Step 3: Coalesced MMIO Handling

Coalesced MMIO ring as contention point
- One ring per-VM
- Readers must synchronize
- Currently protected by qemu_global_mutex

Short-term solution
- Skip flush if target device does not use coalesced MMIO
- Affects VGA and E1000 so far

Long-term solution
- One ring per-device – or MMIO region
- Socket-based ioeventfd may be the answer
Step 4: IRQ Forwarding

Typical IRQ path
- Device changes level / generates edge
- IRQ routers (PCI host, bridges, IRQ remapper, etc.) forward to interrupt controller
- Interrupt controller forwards to CPU
=> Routing involves multiple device models, i.e. potentially multiple critical sections

Cannot take the long road if source & sink are in-kernel
- Hacks exist to explore and monitor routes – on x86
=> Generic mechanism required

Fast path from device to target CPU
- No interaction with routing devices
- State changes (reroutes, blockings) reported to subscribers
- Routing device states can be updated on demand
The Harder Nuts –
Step 5: Concurrent Device Models

Mandatory
- Separate contexts to handle host-originated events
- Enables event prioritization and parallelizing
- iothread(s) can remain “best effort” zone(s)

Variant A
- Per-device lock for atomic sections
- Separate iothreads

Variant B
- Device server thread executes atomic sections
Variant A: A Lock for Every Device

Per-device lock
- Protects atomic sections (PIO/MMIO requests, event processing)
- Can be taken over VCPU or I/O thread contexts

Separate I/O threads
- Process host-triggered work
  - Device-related file descriptor callbacks
  - Bottom-halves
- Granularity: device or group of devices

Downside
- MMIO addresses device, device issues DMA to another device
  => lock nestings, lock recursions, deadlocks
- Which lock to acquire in which order?
- Can we drop the device lock while calling core services?
Variant B: Device Server Thread

Server thread runs device jobs
- Host-triggered work
- Complex guest-triggered work

Guest I/O requests forwarded to server
- Write requests can be synchronous and asynchronous
- Reads must be synchronous

Trivial I/O requests do not require server context
- get/set register without side effects

Thread ensures atomicity of device model
=> no locks required (famous last words...)

Downsides
- May require careful ordering of state changes
- May require use of atomics & barriers
Work in Progress

QEMU activities
- Implement sketched road map
- Currently focusing on variant B
- Primary target
  - E1000 device model
  - KVM with in-kernel IRQ chips

Kernel activities
- Hunt & analyze potential latency spots (hundred µs range)
- Address IRQ thread management issue
Implementation Footnote:
Fun with glibc and POSIX

glibc's condition variables
+ priority inheritance mutexes
= deadlock

Background
- Internal condvar locks aren't PI-aware
- Using PI locks unconditionally considered too heavy
- Lacking POSIX interface to declare PI for condvars
- Patches exist for pthread_condattr_setprotocol_np
- Ignored by glibc folks :-(

Workarounds
- Use priority ceiling
  - Costly (one syscall per mutex lock/unlock)
  - All participating threads must be SCHED_FIFO/RR
- Don't use condvars
Summary

Many benefits of using KVM as RT hypervisor
- Full virtualization feature set
- Matured support for broad range of guests

Restricted RT support so far
- Works well without QEMU in the loop
- User space VM exits trigger huge latencies

Ongoing work to reduce restrictions
- Parallelize and prioritize QEMU device models
- Next goal: emulated RT networking
- Event loop latencies ≪1 ms in reach

Progress on real-time will improve SMP scalability as well!
Any Questions?

Thank you!