Real-Time Virtualization – How Crazy Are We?
Real-Time Systems Can Benefit from Virtualization

Virtualizable real-time systems

• **Possible scenarios**
  • Control systems (industry, healthcare, automotive etc.)
  • Communication systems (media streaming & switching, etc.)
  • Trading systems (stocks, goods, etc.)
  • …

• **Primary drivers**
  • Consolidation, include mixed criticality
  • Legacy system migration
  • [Development & test]
Virtualization ≠ acceleration

The critical data path with and without virtualization

Native real-time setup

Hardware-assisted virtualization

Resource sharing & abstraction via emulation
RT Virtualization – Two Architectural Options

- Real-time Application
- GPOS
- RTOS
- Partitioning Hypervisor
  - CPU
  - I/O

- Real-time Application
- RTOS
- RT-Hyperv.
- Hypervisor
  - CPU
  - I/O

- Guest OS
- Real-Time Host OS
  - CPU
  - I/O
Architecture of a KVM-based RT-Hypervisor

- Non-RT Application
- Real-time Application
- Stand-alone RTOS or AMP Guest
- RT-enhanced QEMU
- PREEMPT-RT + KVM on dedicated cores
- Hardware
PREEMPT-RT enables RT-Virtualization

Role of Linux extension PREEMPT-RT

- **Reduce worst-case event delivery latencies**
- **Integrates KVM support**
  - Original use-case: virtualization + native RT applications
- **Allows to prioritize virtualization workload over uncritical tasks**
- **Can be combined with CPU isolation**
  - 1:1 assignment: host CPU – RT guest CPU
  - Off-load all non-RT tasks (including low-priority QEMU threads)
  - Warning: No 100% guest CPU load feasible!
    - NO_HZ_FULL extensions work toward enabling this
Decent Latencies Achievable in KVM-only Setups

Measuring I/O latency of an RT Guest

- **Host setup**
  - KVM on x86 PREEMPT-RT Linux
  - Virtual machine on dedicated core
  - Intel NIC (E1000 family) as I/O device, directly assigned to guest
  - Permanent disk I/O load

- **Guest setup**
  - Proprietary RTOS
  - Real-time network stack

- **Measurement setup**
  - Linux/Xenomai (native installation)
  - Real-time network stack RTnet
  - Periodic ICMP ping messages sent to target
  - Recorded round-trip latency (error <50 µs)

=> **Worst-case latency after 16h: 330 µs**
RT-QEMU is Required for Emulating in Real-Time

QEMU as a Real-Time Device Emulator

**Scenarios**
- Guest uses NIC A, host has NIC B attached
- Legacy devices are no longer available on a modern host
- Multiple guests share single I/O interface for talking to different devices (e.g. on a CAN bus)

**QEMU can handle such scenarios via emulation**

**Requirements on emulation**
- Equivalent functional behavior
- Devices models need to react in time on guest requests
- Devices models need to deliver external events to the guest timely
Concurrencies in QEMU/KVM –
The Big QEMU Lock (BQL)
Critical BQL Zones

CPUState
- Read/write access
- cpu_single_env

Coalesced MMIO flushing

PIO/MMIO request-to-device dispatching

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)
Challenge 1: Management of Task Priorities

There can be many task involved

- VCPU threads
- VIRQ injection threads (QEMU: iothreads)
- Kernel threads (IRQ, worker, RCU, forgot anything?)

Problems

- Wrong configuration destroys RT
- ...or locks up parts or all of your system
- Actually a generic RT Linux issue

Proposals?

- Tool-based dependency discovery?
- Tool-based configuration?
- (More) automatic configuration?
Challenge 2: Management of IRQ Parameters

Relevant IRQ parameters
- CPU affinity
- Thread priority (if any)

How to configure in advance?
- Line-basedIRQs may be reachable via /proc/irq
- MSIs are not...
- Dynamic IRQ numbers – how to associate with devices?

Proposals?
- Something like /sys/devices/.../<device>/irq_vector<N>/...?
Safe Isolation via Linux?

- Code size
- Certification
- Consolidation
- Mixed Criticality
- ASIL
- Formal Methods
- SIL2Linux
- Legacy Code
- Validation
- Multicore

Stahlkocher, CC BY-SA 3.0
What about postponing the hypervisor start?

Basic concept of late partitioning

1. Boot phase
2. Partitioning phase
3. Operational phase
Jailhouse: Keep it simple, keep it open

The Philosophy of Jailhouse

- Avoid emulation, focus on hardware assisted isolation
  - No overcommitment, no scheduler, static partitioning
  - Directly assign physical devices, do not emulate them
  - You need more? Use KVM!
- Only expose resources that are required for operation
  - No boot-up phase virtualization
  - Board initialization done by Linux
- Off-load uncritical tasks to Linux
  - Initial setup / image loading
  - Reconfigurations while in non-operational mode
  - Monitoring, logging etc.
- Released under GPLv2

=> Minimal-sized certifiable hypervisor with full CPU assignment and Linux look-and-feel