ORACLE®

Experience in using Qemu/KVM as a tool to develop software for a complex new hardware device

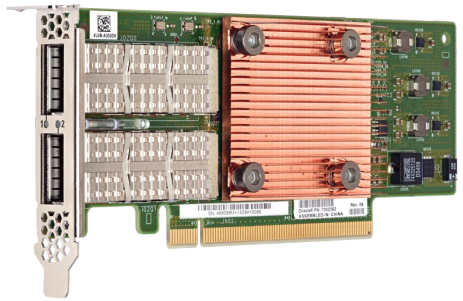
Knut Omang, KVM Forum/LinuxCon Aug 24, 2016

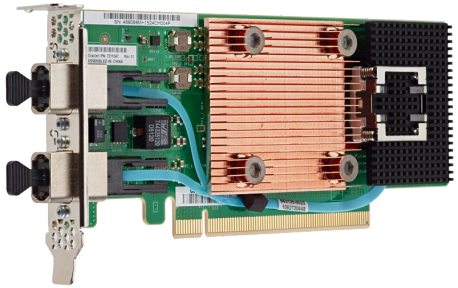
Agenda

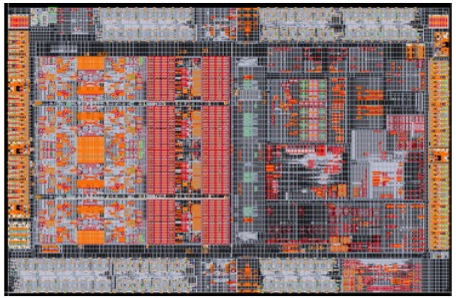
- About our development target
- Our goals - why virtualization and QEMU/KVM?
- A taste of the tool set developed and some use cases
- What we achieved
- Challenges and ideas for further work

Device: Oracle Infiniband (IB) HCA

- Oracle's first “in-house” Infiniband HCA
  - Highly asynchronous usage model
  - SR/IOV support w/integrated virtual switches
  - Integrated subnet management agent
  - On-chip MMU compatible with CPU page tables
  - NIC offloads for Ethernet over IB and IP over IB







Infiniband (IB): Defined by software and hardware

- HCA: Host Channel Adapter (= IB network adapter)
  - Defines set of operations to support
  - does not define if hardware or software implementation
  - Standard defines verbs semantics, requirements, not syntax/implementation
- Linux implementation:
  - RDMA (= Remote Direct Memory Access) support
  - Kernel and user space (libibverbs)
  - Driver entry point support at kernel *and* user level
- IB standard counts ~2500 pages…
  - but still details left to implementations..

The Oracle IB HCA SW opportunity

- Participate in making something new and cool
- Start from clean sheets - little baggage - do it right!
- SW effort started early enough to influence HW!
- The adventure!

"...far, far away he could see something light and shimmering..."

Good tools half the work..

or

- and better quality results too!

Some software team “survival rules”

- Time set aside to develop tools a fundamental success criteria
  - “Yes, a great idea, as long as it doesn't take any time...”
- Write usable target software from the start
  - Driver, firmware and library code should be as little affected by lack of hardware as possible!
  - Make tests early as valuable as possible also later in project
- Test driven development whenever practically possible
  - Not considered fully tested until part of regression testing:
    - “I tested it a few days ago and it worked then...”
  - Nobody is excempt from writing and maintaining test, test code and -tools.
  - Test/verification related work probably more than 95% of the whole combined software task
- Continuous integration
  - Check-in testing + nightly

Motivation for using virtualization

- Allow target software and firmware development to start early
- Minimize the impact of “test only” code
- Understand (and play with) the new functional options of PCIe (vs PCI)
- Find bugs/weaknesses/design flaws in hardware before tapeout
- Reduce SW team time on critical path = minimize pain..
- Get more testing out of the same hardware

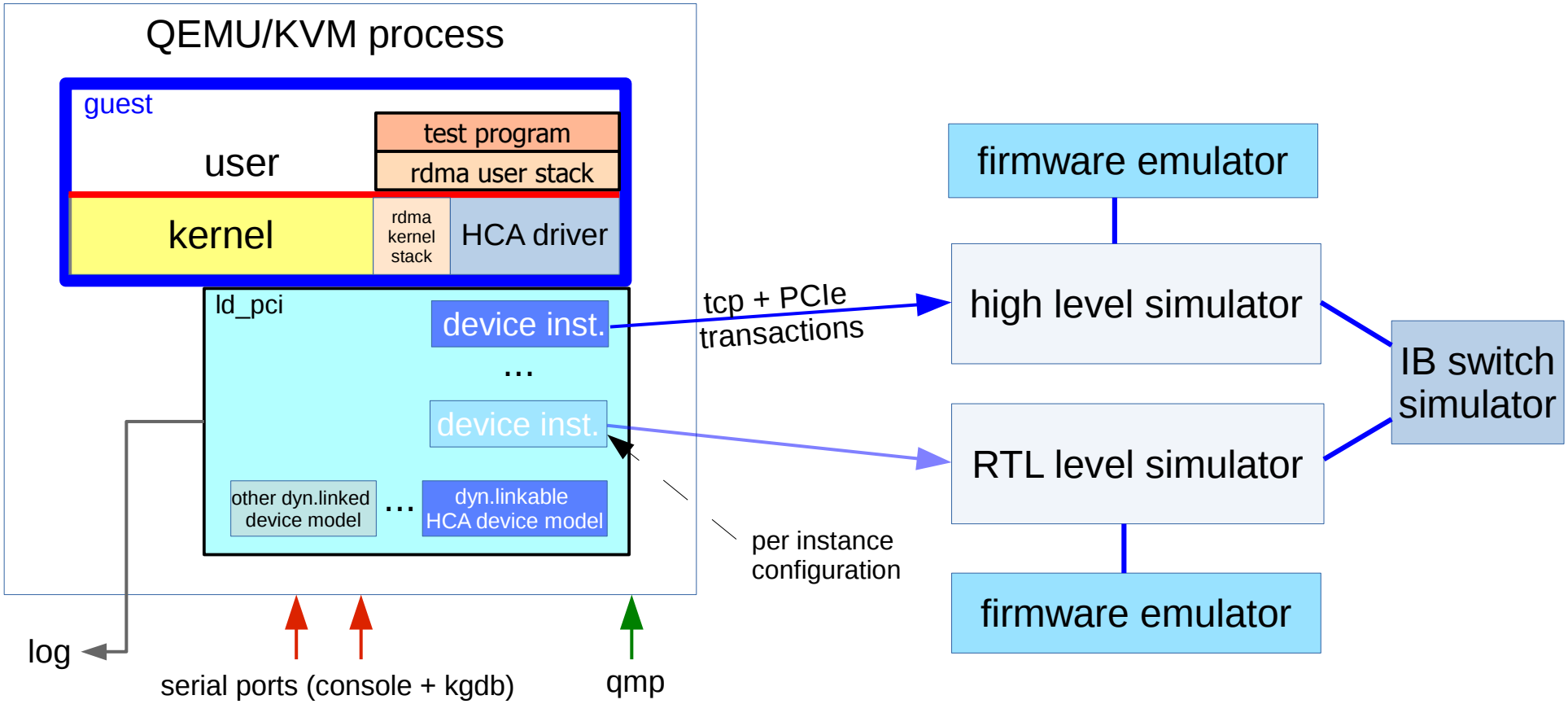
Why QEMU/KVM?

- Source access - needed to be able to extend/modify
- Shortest path to get something useful to demonstrate
  - I had already used it in a previous project
- Excellent hypervisor environment = a full Linux!
- PCIe support => Q35 … bleeding edge!
- Nested virtualization (device assignment, SR/IOV, Xen target OS)
- The wonderful COW support in qcow2!
- A good and active community

Additional challenge factors

- Wanted flexible solution for current and future simulation/emulation
  - A high level device model under development
  - Access to RTL under heavy evolvement - could we automate and interface?
  - Other simulation/emulation models? Be prepared for the next step
  - Make the system flexible enough to be useful to other hardware projects
- Limited hardware resources available for software testing
  - Old servers, limited memory, 60G disks..
- Little support resources available
  - needed a solution simple to deploy and manage
  - avoid reinventing wheels - rely on existing tools if possible/achievable
- QEMU vs Virtualbox vs Xen (Oracle VM)

The abstract simulated Oracle Infiniband HCA



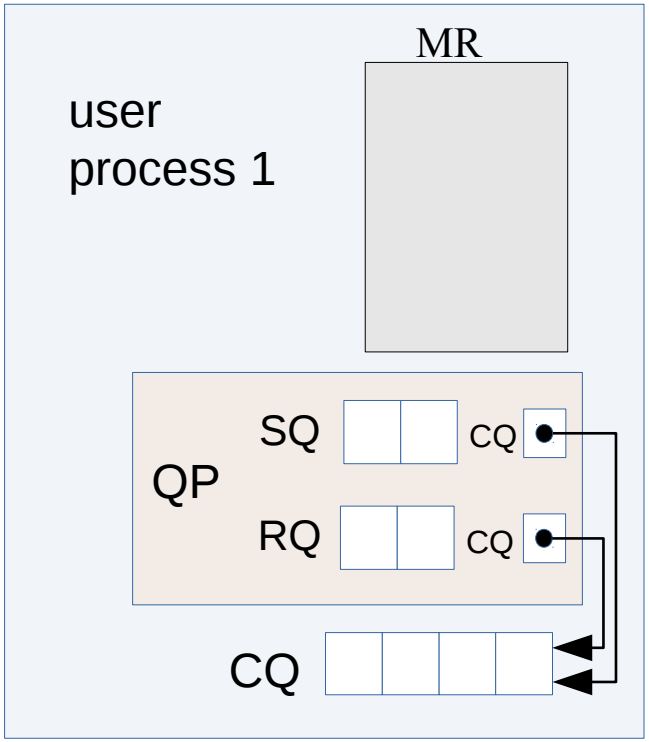
ld_pci: dynamically loadable PCI device support

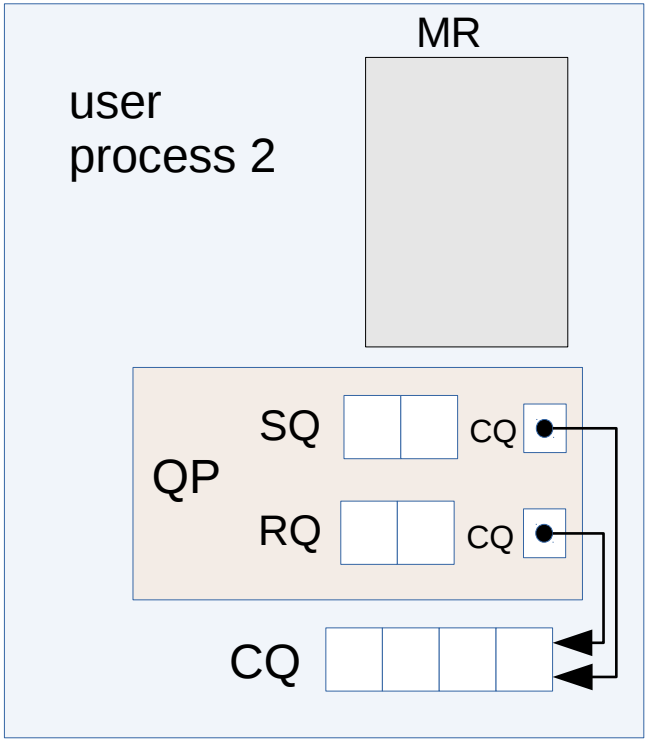
- Plugin framework
- single API header file, minimal include deps
  - < 10 callbacks/values implemented by device model, incl. API version info
  - ~30 utility functions implemented by VMM
    - implementations by QEMU patch set, [VirtualBox], kernel unit test framework, …?
  - single qemu patch: 16 files changed, 1069 insertions(+), 4 deletions
- Callbacks to converge towards QEMU API
  - goal to allow models to be compiled “directly” against QEMU
  - + “lift out” existing devices..
- Recent impl for QEMU benefits increasingly from:
  - hotplug support improvements
  - properties and the QEMU object model

ld_pci plugin for the Oracle Infiniband HCA

- PCIe enumeration - device realization:
  - path and name property
    - QEMU tries to look up device type - if it fails, asks ld_pci. Device specified as eg.
      `-device sif,path=/var/lib/kvmrun/sif/o4kvm170,id=sif0,bus=pcie_port.0`
      Triggers dlopen if necessary (ref.cnt): Each .so adds n types to QEMU type definitions
  - device instance create
    - read configuration file w/ip:port
    - try to connect to simulator
    - fallback implementation if no simulator instance listening
- Device unrealize:
  - disconnect from simulator if necessary
  - delete device instance
  - deref, if ref.cnt == 0: Remove owned types from type definitions, dlclose()

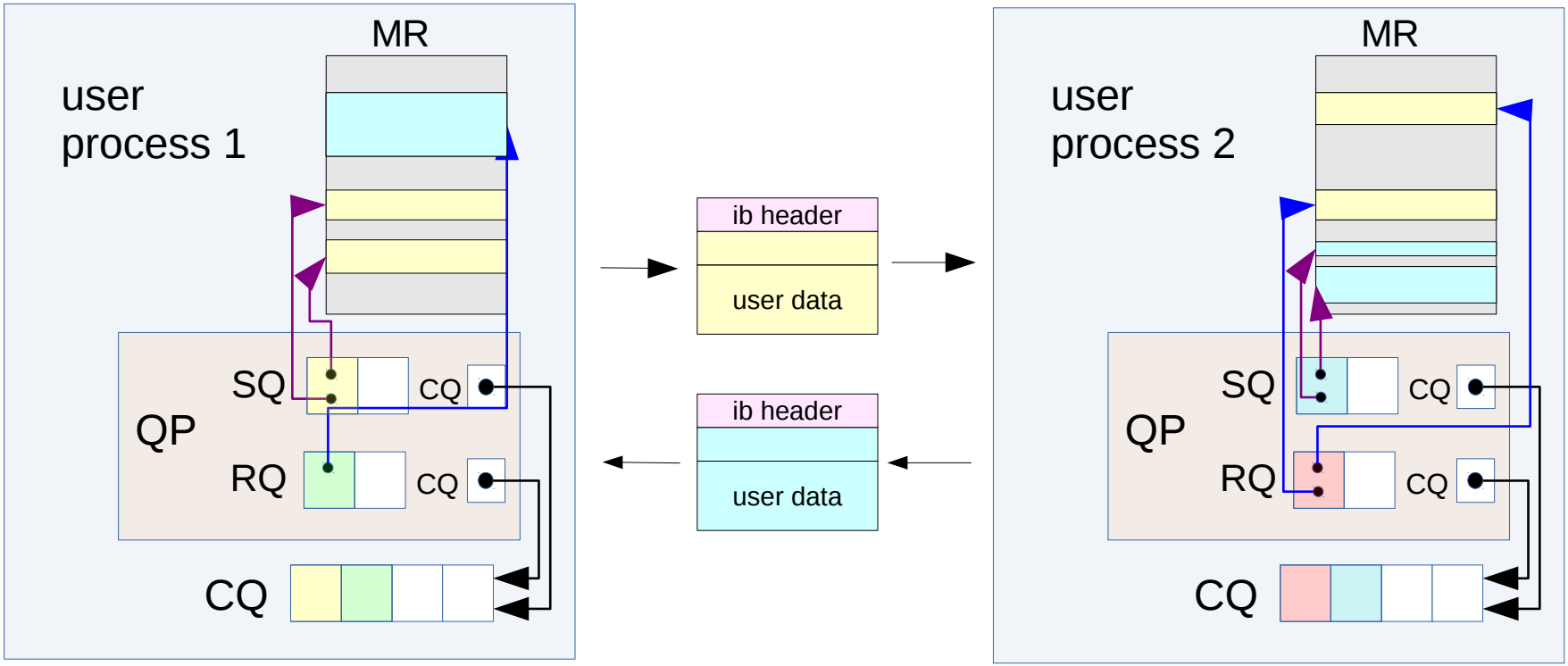
Infiniband communication example





ibv_reg_mr() register memory

Infiniband communication example



ib_post_send() receive a message

Modeling the PCIe side of the Oracle Infiniband HCA

- PCIE config space
  - 3 BARs: FW access BAR, MSIX BAR, WR posting BAR (“Collect buffers”)
  - SR/IOV capability and support
  - Various other PCIe capabilities
- MSIX support for interrupts
- (Lots of) DMA
  - triggered by WRs
  - implicit by device
- Initial impl: Use QEMU support
- As simulation support matured: Move functions over..
  - but keep/support fallback!

Key issues for efficient driver development

- Development cycle time
  - too long: Have to multiplex to be efficient - context switch (cost? what is too long?)
  - boot time (servers… , init vs systemd, …) [power-on to ssh access server: 3 min, VM: 30 secs]
  - non-fatal error...reload cycle time [hotplug device on VM: 11 sec]
- Observability
  - IB: A lot is handled by hardware, at very high speed
  - Hard to observe without affecting output
- Debugging facilities + interactive when possible/necessary
- Test coverage
  - what do we dare to change?
  - huge test space: Infiniband, OFED/rdma impl, MLX “compliance”, ULP/appl quirks, OS distro, OS versions, error scenarios,…
  - initial focus on “safety net”
  - Can't write all tests multiple times - they have to work in all environments!

Observability: Debugging communication flow

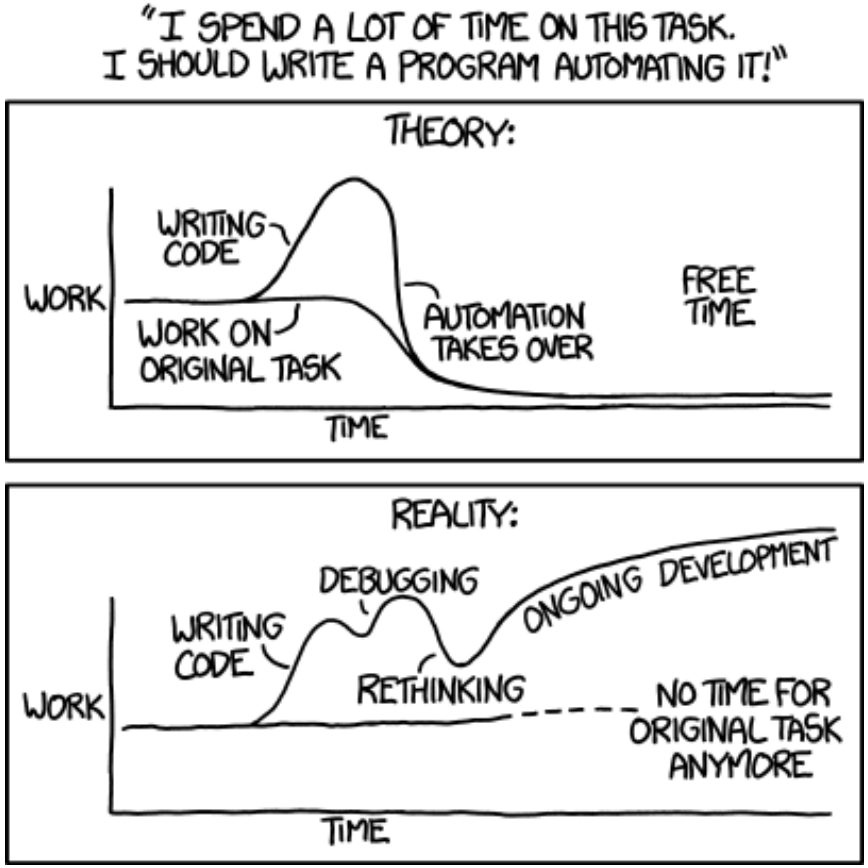
- Hardware: relatively black box
  - Some debugging facilities in hardware, but limited use cases
  - can time share on PCIe tracer (heavily contended resource, very expensive, lab work to set up)
  - limited memory in tracer, heavy user interface
  - TBD: Parsing output + wireshark support...
- Simulation: Full insight at all levels
  - driver logging, ftrace etc
  - Qemu plugin level symbolic packet snooping
  - Simulator frontend logging
  - RTL simulation: Waveform output - accurate but painfully slow

Testing - safety net: Controlling DMA access to memory

- Catch early dev bugs before they lead to random memory errors:
  - driver bug calculates invalid address in request to HCA
  - detect hardware/simulator errors or wrong/unexpected usage from driver
- Initial version: Implement driver DMA API
  - Communicate via CSRs in simulation only BAR page
  - simple “iommu” support in Qemu device model to trap and fault on inappropriate accesses
- Virtual IOMMU support
  - Initial patch set
  - GSOC project => success thanks to Jan Kiszka!

Debugging facilities: symbolic interactive kernel debugging

- script to load driver while saving symbol info to NFS
- gdb + load symbol defs
- kgdb enabled guest kernel
- 2 serial ports to separate console from kgdb
- Handle NULL pointer exception/fatal crash during driver load
  - boot with 0 devices
  - load driver → save symbol info to NFS
  - hot plug a new device → panic, but with gdb symbolic debugging support



https://xkcd.com/1319/

and

http://events.linuxfoundation.org/sites/events/files/slides/LinuxCon2015-Intel.pdf

Managing 100's of (almost identical) VMs

- disk space, no time for management/training/docs
- enter kvmrun: shell script + config + command line + rpm support
  - templates using qcow2 backing files
  - start <vm list>, stop <vm list>
  - simple numbering scheme for names, ethernet addresses, port assignment
  - cfg feature creep: pci or pcie, #of root ports, vlan support, multiple subnets, bridges, disk interface, ethernet model, memory size, grub, snapshots, kickstart, passthrough, vfio, auth. setup… (sigh..)
- Reinvented wheel?
  - Looked at libvirt etc...
  - oVirt, …?
  - At the time: Immature, needed more control, had little time => future work…
  - Not to mention brain stretch...

Deployment today

- Continuous integration based on Jenkins and Gerrit
- Every commit (several git projects) subject to 2-stage test
  - Smoke tests mostly in simulated environments (parallel on multiple commits)
  - Serialized checkin regression tests (hw and simulation)
- Nightly long regression test set
- Test base:
  - Gtest based tests (unit tests + simple system tests)
  - Standard RDMA test applications
  - Several other stress tests
- Not achieavable with same amount of HW without VMs

Summary: What did we achieve?

- Ability to start very early in the hardware development
- Found hardware bugs/minor design issues (before tapeout!)
- Significantly shorter development cycle both for firmware and software
- Had working driver (and firmware) when first hardware arrived
- Longer term gains: Toolbox and infrastructure reusable for future projects

ORACLE

What more could we have done?

- A lot - the sky is the limit!
- Resource allocation issue
- Trust, competence - a lot can go wrong too!

Some technical challenges

- Churn in QEMU, but for the good ;-)
  - cleanup handling, ongoing refactoring, understanding: multithreading issues, object models..
- Memory consistency over PCIe
  - write ordering from single initiator
- Rebase while keeping team and CI system happy…
  - .rpms, testing the test system, testing the test system tests….
- Maintenance of 100's of almost identical VMs…
  - use case fairly different from “normal” VM usages..!

The timeout scaling problem

Response times:

- Hardware: nanosecond range
- High level simulation: millisecond range
- Full RTL simulation in software: 10s of seconds range

Problem: Timeouts!

- Partial solution for rdma stack: patch to add scaling factor parameter
- Filtering system log :-(
- Network stack?
- Core kernel: soft lockups, NMIs,...
- Can QEMU help?

Further QEMU related work

- The timeout scaling problem
- Better tools to “underneath” the VM
  - stopping the OS of a virtual machine - ^C equivalent to get to kgdb
  - have 'echo g > /proc/sysrq-trigger' but only if system is live
  - (right now: identify a trigger, call kgdb_breakpoint(), recompile, hotplug..)
- Migrate from huge shell script VMMS to oVirt, virt-manager
- QMP improvements/extensions...
- Get my SR/IOV patches merged :-)
- New PCIe features such as PRI (Page Request Interface), …
- Ideas?

Changing peoples minds..

Some challenges mostly in the “social engineering” category:

- Long term vs short term!
- Early deadlines - management need for tracking (= trust)
- Minimize + justify recurring overhead
- Make good enough arguments for competence diversity

- The evolutionary approach…!

“Neat” – also for developers?

Personal takeaway

- If you want to understand how some piece of hardware works, try to write an emulation for it!

- Questions?