QEMU Hotplug Infrastructure and Implementing PCI Hotplug for PowerKVM

Michael Roth
mdroth@linux.vnet.ibm.com
Uses for PCI Hotplug in Physical Machines

- Add/remove adaptors
- Serviceability
- upgrading to newer hardware (features/performance)
- Expandability
  - NIC for faster/secondary 10G
  - HBA for storage array
  - Even more important for virtual machines
Basic requirements for PCI hot plug

- Plug in device (user-initiated)
- Notify OS (user-initiated)
- Enable/probe/configure device
- Signal completion
Basic requirements for PCI hot unplug

- Notify OS (user-initiated)
- Unconfigure/disable device
- Signal completion
- Unplug device
QEMU “user” Interface

- Initiated by QEMU management interfaces (HMP/QMP)
- `device_add virtio-net-pci,bus=pci.0,id=hp0`
- `device_del hp0`
QEMU hotplug hooks

- `qbus_set_hotplug_handler(BusState *bus, DeviceState *handler)`
- `handler->plug(bus, pcidev)`
- `handler->request_unplug(bus, pcidev)`
- `handler->unplug(bus, pcidev)`
- Behavior depends on hotplug handler/platform/architecture
PCI Hotplug Device Models in QEMU

- ACPI (Advanced Configuration and Power Interface)
- SHPC (Standard Hot Plug Controller)
- PCIe native hotplug
- sPAPR/pSeries Dynamic Reconfiguration
ACPI-based hotplug

- Firmware interface for device configuration and power management
- Uses set of tables to provide device descriptions and platform code/methods to interact with hardware
- Supports both PCI and PCIe hotplug (memory/cpu too)
- Superseded SHPC and PCIe native
- Supported for x86 i440fx
ACPI-based hotplug – basic workflow

- “acpi-gpe0” status/enabled registers for event notifications
- SCI interrupt to signal OS
- “acpi-pci-hotplug” registers for slot up/down/eject

```
device_add → handler->plug():
    "acpi-pci-hotplug".up |= 1 << slot
    "acpi-gpe0".status |= ACPI_PCI_HOTPLUG_STATUS
    SCI interrupt
    OS brings device online
```
ACPI-based hotplug – basic workflow

- “acpi-gpe0” status/enabled registers for event notifications
- SCI interrupt to signal OS
- “acpi-pci-hotplug” registers for slot up/down/eject

device_del $\rightarrow$ handler->request_unplug():
  - “acpi-pci-hotplug”.down |= 1 << slot
  - “acpi-gpe0”.status |= ACPI_PCI_HOTPLUG_STATUS
  - SCI interrupt
  - “acpi-pci-hotplug”.eject |= 1 << slot
  - QEMU cleans up and finalizes device
SHPC-based hotplug

- PCI-SIG spec, newer than ACPI
- Defines both OS-facing and user-facing interfaces
- PCI only (PCIe has native support), PCIe-to-PCI bridge?
- Supports host bridges and pci-to-pci bridges
- In QEMU, only supported for pci-to-pci bridges
- Usable by... any pci host?
SHPC-based hotplug – basic workflow

- PCI_CAP_ID_SHPC advertised via PCI capabilities
- slot select register
- slot operation register: attention/power indicators, slot on/off/enable

`device_add → handler->plug():`

- “close” MRL
- “push” attention button
- SHPC sends OS interrupt
- OS checks that MRL is secured, card present, no power
- OS powers on and enables device, sets LED
SHPC-based hotplug – basic workflow

- PCI_CAP_ID_SHPC advertised via PCI capabilities
- slot select register
- slot operation register: attention/power indicators, slot on/off/enable

`device_del → handler->request_unplug()`:  
“push” attention button
SHPC sends OS interrupt
OS unconfigures device, powers off device, sets LED
PCIe native hotplug

- PCI-SIG spec, built into PCIe standard
- Similar to SHPC, port capability instead of bridge
- Supported via PCIe root/downstream port for x86 'q35', and ARM 'virt' (in theory)
- Little bit more setup (no hotplug to internal host bus):

```bash
qemu -M q35 \
-device ioh3420,multifunction=on,bus=pcie.0,id=port9-0,addr=9.0,chassis=0 \
-device ioh3420,multifunction=on,bus=pcie.0,id=port9-1,addr=9.1,chassis=1
```
PCIe native hotplug

- PCI_CAP_ID_EXP advertised via PCI cap list
- PCIe cap structure already includes registers for slot management: slot capabilities/control/status registers
- Available for root/downstream ports with a slot associated (as opposed to ports that link up internal devices)

- Same basic workflow as SHPC, except each slot is a bridge with its own “SHPC” (still needed PCIe-specific drivers)
sPAPR/pSeries Dynamic Reconfiguration

- PCI-SIG spec, built into PCIe standard
- Firmware interface for device configuration and power management
- Uses a set of table to provide device descriptions like ACPI
- Unlike ACPI, uses RTAS for executing platform code
- Supported for -M spapr
- Supports memory/CPU/PHB as well
sPAPR/pSeries Dynamic Reconfiguration

- Each hotpluggable resource/slot has a DR Connector
- Described via Open Firmware Device Trees:
  - PCI DRC for slot 1:
    - drc-indexes[7]: 0x40000008
    - drc-types[7]: SPAPR_DR_CONNECTOR_TYPE_PCI
    - drc-names[7]: "C8"
    - drc-power-domains[7]: -1 (auto power)
    - EPOW interrupts/notifications
sPAPR/pSeries Dynamic Reconfiguration

- RTAS for OS<->fw interaction
- rtas-set-indicator: set LEDs/isolation/allocation states
- rtas-configure-connector: fetch device trees for plugged devices
- rtas-get-sensor-state: entity sense
- rtas-{set,get}-power-domain: power on/off devices
sPAPR/pSeries Dynamic Reconfiguration

- Basic workflow:

  device_add $\rightarrow$ handler->plug():
  
  Set DRC isolation state to unisolated (OS-visible)
  Generate hotplug event with DRC index for slot
  EPOW interrupt to notify OS
  OS fetches device tree via configure-connector
  OS sets indicator LED to signal completion
sPAPR/pSeries Dynamic Reconfiguration

- Basic workflow:

  `device_add` → `handler->plug()`:
  - Set DRC isolation state to unisolated (OS-visible)
  - Generate hotplug event with DRC index for slot
  - EPOW interrupt to notify OS
  - OS fetches device tree via `configure-connector`
  - OS sets indicator LED to signal completion
Questions?
Other items

- PHB Hotplug