PCI Pass-through in Xen ARM

manish.jaggi@caviumnetworks.com

Draft-3

Introduction

This document describes the design for the PCI passthrough support in Xen ARM.
The target system is an ARM 64bit Soc with GICv3 and SMMU v2 and PCIe devices.

Revision History

Changes from Draft-1:

a) map_mmio hypercall removed from earlier draft
b) device bar mapping into guest not 1:1
c) holes in guest address space 32bit / 64bit for MMIO virtual BARs
d) xenstore device's BAR info addition.

Changes from Draft-2:

a) DomU boot information updated with boot-time device assignment and hotplug.
b) SMMU description added
c) Mapping between streamID - bdf - deviceID.
d) assign_device hypercall to include virtual(guest) sbdf.
Toolstack to generate guest sbdf rather than pciback.

Index

1. Background of PCI passthrough

Passthrough refers to assigning a pci device to a guest domain (domU) such that
the guest has full control over the device. The MMIO space and interrupts are
managed by the guest itself, close to how a bare kernel manages a device.

Device's access to guest address space needs to be isolated and protected. SMMU (System MMU - IOMMU in ARM) is programmed by xen hypervisor to allow device access guest memory for data transfer and sending MSI/X interrupts. PCI devices generated message signalled interrupt write are within guest address spaces which are also translated using SMMU.
For this reason the GITS (ITS address space) Interrupt Translation Register space is mapped in the guest address space.

2. Basic PCI Support for ARM

The apis to read write from pci configuration space are based on segment:bdf. How the sbdf is mapped to a physical address is under the realm of the pci host controller.

ARM PCI support in Xen, introduces pci host controller similar to what exists in Linux. Each drivers registers callbacks, which are invoked on matching the compatible property in pci device tree node.

2.1 pci_hostbridge and pci_hostbridge_ops

The init function in the pci host driver calls to register hostbridge callbacks:

```
int pci_hostbridge_register(pci_hostbridge_t *pcihb);

struct pci_hostbridge_ops {
u32 (*pci_conf_read)(struct pci_hostbridge*, u32 bus, u32 devfn,
u32 reg, u32 bytes);
void (*pci_conf_write)(struct pci_hostbridge*, u32 bus, u32 devfn,
u32 reg, u32 bytes, u32 val);
};

struct pci_hostbridge{
u32 segno;
paddr_t cfg_base;
paddr_t cfg_size;
struct dt_device_node *dt_node;
struct pci_hostbridge_ops ops;
struct list_head list;
};
```

A pci conf read function would internally be as follows:

```
u32 pcihb_conf_read(u32 seg, u32 bus, u32 devfn,u32 reg, u32 bytes)
{
pci_hostbridge_t *pcihb;
list_for_each_entry(pcihb, &pci_hostbridge_list, list)
{
if(pcihb->segno == seg)
return pcihb->ops.pci_conf_read(pcihb, bus, devfn, reg, bytes);
}
return -1;
}
```

2.2 PHYSDEVOP_pci_host_bridge_add hypercall

Xen code accesses PCI configuration space based on the sbdf received from the guest. The order in which the pci device tree node appear may not be the same order of device enumeration in dom0. Thus there needs to be a mechanism to bind the segment number assigned by dom0 to the pci host controller. The hypercall is introduced:

```
#define PHYSDEVOP_pci_host_bridge_add 44
struct physdev_pci_host_bridge_add {
/* IN */
uint16_t seg;
uint64_t cfg_base;
uint64_t cfg_size;
};
```

This hypercall is invoked before dom0 invokes the PHYSDEVOP_pci_device_add hypercall. The handler code invokes to update segment number in pci_hostbridge:

```
int pci_hostbridge_setup(uint32_t segno, uint64_t cfg_base, uint64_t cfg_size);
```

Subsequent calls to pci_conf_read/write are completed by the pci_hostbridge_ops
of the respective pci_hostbridge.

2.3 Helper Functions

a) pci_hostbridge_dt_node(pdev->seg);
Returns the device tree node pointer of the pci node from which the pdev got
enumerated.

3. SMMU programming

3.1. Additions for PCI Passthrough

3.1.1 - add_device in iommu_ops is implemented.

This is called when PHYSDEVOP_pci_add_device is called from dom0.

```
.add_device = arm_smmu_add_dom0_dev,
static int arm_smmu_add_dom0_dev(u8 devfn, struct device *dev)
{
if (dev_is_pci(dev)) {
struct pci_dev *pdev = to_pci_dev(dev);
return arm_smmu_assign_dev(pdev->domain, devfn, dev);
}
return -1;
}
```

3.1.2 dev_get_dev_node is modified for pci devices.

The function is modified to return the dt_node of the pci hostbridge from
the device tree. This is required as non-dt devices need a way to find on
which smmu they are attached.

```
static struct arm_smmu_device *find_smmu_for_device(struct device *dev)
{
struct device_node *dev_node = dev_get_dev_node(dev);
....

static struct device_node *dev_get_dev_node(struct device *dev)
{
if (dev_is_pci(dev)) {
struct pci_dev *pdev = to_pci_dev(dev);
return pci_hostbridge_dt_node(pdev->seg);
}
...
```

3.2. Mapping between streamID - deviceID - pci sbdf - requesterID

For a simpler case all should be equal to BDF. But there are some devices that
use the wrong requester ID for DMA transactions. Linux kernel has pci quirks
for these. How the same be implemented in Xen or a diffrent approach has to be
taken is TODO here.
Till that time, for basic implementation it is assumed that all are equal to BDF.

4. Assignment of PCI device

4.1 Dom0

All PCI devices are assigned to dom0 unless hidden by pci-hide bootargs in dom0.
Dom0 enumerates the PCI devices. For each device the MMIO space has to be mapped
in the Stage2 translation for dom0. For dom0 xen maps the ranges from dt pci
nodes in stage 2 translation during boot.

4.1.1 Stage 2 Mapping of GITS_ITRANSLATER space (64k)

GITS_ITRANSLATER space (64k) must be programmed in Stage2 translation so that SMMU
can translate MSI(x) from the device using the page table of the domain.

```

4.1.1.1 For Dom0
-----------------
GITS_ITRANSLATER address space is mapped 1:1 during dom0 boot. For dom0 this
mapping is done in the vgic driver. For domU the mapping is done by toolstack.

4.1.1.2 For DomU
-----------------
For domU, while creating the domain, the toolstack reads the IPA from the
macro GITS_ITRANSLATER_SPACE from xen/include/public/arch-arm.h. The PA is
read from a new hypercall which returns the PA of the GITS_ITRANSLATER_SPACE.
Subsequently the toolstack sends a hypercall to create a stage 2 mapping.

Hypercall Details: XEN_DOMCTL_get_itranslater_space

/* XEN_DOMCTL_get_itranslater_space */
struct xen_domctl_get_itranslater_space {
/* OUT variables. */
uint64_aligned_t start_addr;
uint64_aligned_t size;
};
typedef struct xen_domctl_get_itranslater_space xen_domctl_get_itranslater_space;
DEFINE_XEN_GUEST_HANDLE(xen_domctl_get_itranslater_space;

4.2 DomU
------------
There are two ways a device is assigned
In the flow of pci-attach device, the toolstack will read the pci configuration
space BAR registers. The toolstack has the guest memory map and the information
of the MMIO holes.

When the first pci device is assigned to domU, toolstack allocates a virtual
BAR region from the MMIO hole area. toolstack then sends domctl
xc_domain_memory_mapping to map in stage2 translation.

4.2.1 Reserved Areas in guest memory space
--------------------------------------------
Parts of the guest address space is reserved for mapping assigned pci device's
BAR regions. Toolstack is responsible for allocating ranges from this area and
creating stage 2 mapping for the domain.

/* For 32bit */
GUEST_MMIO_BAR_BASE_32, GUEST_MMIO_BAR_SIZE_32

/* For 64bit */

GUEST_MMIO_BAR_BASE_64, GUEST_MMIO_BAR_SIZE_64

Note: For 64bit systems, PCI BAR regions should be mapped from
GUEST_MMIO_BAR_BASE_64.

IPA is allocated from the {GUEST_MMIO_BAR_BASE_64, GUEST_MMIO_BAR_SIZE_64}
range and PA is the values read from the BAR registers.

4.2.2 New entries in xenstore for device BARs
-----------------------------------------------
toolstack also updates the xenstore information for the device
(virtualbar:physical bar).This information is read by xenpciback and returned
to the pcifront driver configuration space reads for BAR.

Entries created are as follows:
/local/domain/0/backend/pci/1/0
vdev-N
BDF = ""
BAR-0-IPA = ""
BAR-0-PA = ""
BAR-0-SIZE = ""
...
BAR-M-IPA = ""
BAR-M-PA = ""
BAR-M-SIZE = ""

```

Note: Is BAR M SIZE is 0, it is not a valied entry.

4.2.4 Hypercall Modification for bdf mapping notification to xen

Guest devfn generation currently done by xen-pciback to be done by toolstack only. Guest devfn is generated at the time of domain creation (if pci devices are specified in cfg file) or using xl pci-attach call.

5. DomU FrontEnd Bus Changes

5.1 Change in Linux PCI ForntEnd - backend driver for MSI/X programming

FrontEnd backend communication for MSI is removed in XEN ARM. It would be handled by the gic-its driver in guest kernel and trapped in xen.

5.2 Frontend bus and interrupt parent vITS

On the Pci frontend bus msi-parent gicv3-its is added. As there is a single virtual its for a domU, as there is only a single virtual pci bus in domU. This ensures that the config_msi calls are handled by the gicv3 its driver in domU kernel and not utilising frontend-backend communication between dom0-domU.

It is required to have a gicv3-its node in guest device tree.

6. NUMA domU and vITS

a) On NUMA systems domU still have a single its node.
b) How can xen identify the ITS on which a device is connected.
- Using segment number query using api which gives pci host controllers device node

```
struct dt_device_node* pci_hostbridge_dt_node(uint32_t segno)
```

c) Query the interrupt parent of the pci device node to find out the its.