PCI Express I/O Virtualization Resource on Powerenv

Wei Yang <weiyang@linux.vnet.ibm.com>

Benjamin Herrenschmidt <benh@au1.ibm.com>

Bjorn Helgaas <bhelgaas@google.com>

26 Aug 2014

This document describes the requirement from hardware for PCI MMIO resource sizing and assignment on PowerKVM and how generic PCI code handles this requirement. The first two sections describe the concepts of Partitionable Endpoints and the implementation on P8 (IODA2). The next two sections talks about considerations on enabling SRIOV on IODA2.

1. Introduction to Partitionable Endpoints

A Partitionable Endpoint (PE) is a way to group the various resources associated with a device or a set of devices to provide isolation between partitions (i.e., filtering of DMA, MSIs etc.) and to provide a mechanism to freeze a device that is causing errors in order to limit the possibility of propagation of bad data.

There is thus, in HW, a table of PE states that contains a pair of "frozen" state bits (one for MMIO and one for DMA, they get set together but can be cleared independently) for each PE.

When a PE is frozen, all stores in any direction are dropped and all loads return all 1's value. MSIs are also blocked. There's a bit more state that captures things like the details of the error that caused the freeze etc., but that's not critical.

The interesting part is how the various PCIe transactions (MMIO, DMA, ...) are matched to their corresponding PEs.

The following section provides a rough description of what we have on P8 (IODA2). Keep in mind that this is all per PHB (PCI host bridge). Each PHB is a completely separate HW entity that replicates the entire logic, so has its own set of PEs, etc.

2. Implementation of Partitionable Endpoints on P8 (IODA2)

P8 supports up to 256 Partitionable Endpoints per PHB.

- Inbound

  For DMA, MSIs and inbound PCIe error messages, we have a table (in memory but accessed in HW by the chip) that provides a direct correspondence between a PCIe RID (bus/dev/fn) with a PE number. We call this the RTT.

  - For DMA we then provide an entire address space for each PE that can contain two "windows", depending on the value of PCI address bit 59. Each window can be configured to be remapped via a "TCE table" (IOMMU translation table), which has various configurable characteristics not described here.
  - For MSIs, we have two windows in the address space (one at the top of the 32-bit space and one much higher) which, via a combination of the address and MSI value, will result in one of the 2048 interrupts per bridge being triggered. There's a PE# in the interrupt controller descriptor table as well which is compared with the PE# obtained from the RTT to "authorize" the device to emit that specific interrupt.
  - Error messages just use the RTT.

- Outbound. That's where the tricky part is.

  Like other PCI host bridges, the Power8 IODA2 PHB supports "windows" from the CPU address space to the PCI address space. There is one M32 window and sixteen M64 windows. They have different characteristics. First what they have in common: they forward a configurable portion of the CPU address space to the PCIe bus and must be naturally aligned power of two in size. The rest is different:

  - The M32 window:
    - Is limited to 4GB in size.
    - Drops the top bits of the address (above the size) and replaces them with a configurable value. This is typically used to generate 32-bit PCIe accesses. We configure that window at boot from FW and don't touch it from Linux; it's usually set to forward a 2GB portion of address space from the CPU to PCIe 0x8000_0000..0xffff_ffff. (Note: The top 64KB are actually reserved for MSIs but this is not a problem at this point; we just need to ensure Linux doesn't assign anything there, the M32 logic ignores that however and will forward in that space if we try).
    - It is divided into 256 segments of equal size. A table in the chip maps each segment to a PE#. That allows portions of the MMIO space to be assigned to PEs on a segment granularity. For a 2GB window, the segment granularity is 2GB/256 = 8MB.

  Now, this is the "main" window we use in Linux today (excluding SR-IOV). We basically use the trick of forcing the bridge MMIO windows onto a segment alignment/granularity so that the space behind a bridge can be assigned to a PE.

Ideally we would like to be able to have individual functions in PEs but that would mean using a completely different address allocation scheme where individual function BARs can be "grouped" to fit in one or more segments.

- The M64 windows:
  - Must be at least 256MB in size.
  - Do not translate addresses (the address on PCIe is the same as the address on the PowerBus). There is a way to also set the top 14 bits which are not conveyed by PowerBus but we don't use this.
  - Can be configured to be segmented. When not segmented, we can specify the PE# for the entire window. When segmented, a window has 256 segments; however, there is no table for mapping a segment to a PE#. The segment number *is* the PE#.
  - Support overlaps. If an address is covered by multiple windows, there's a defined ordering for which window applies.

We have code (fairly new compared to the M32 stuff) that exploits that for large BARs in 64-bit space:

We configure an M64 window to cover the entire region of address space that has been assigned by FW for the PHB (about 64GB, ignore the space for the M32, it comes out of a different "reserve"). We configure it as segmented.

Then we do the same thing as with M32, using the bridge alignment trick, to match to those giant segments.

Since we cannot remap, we have two additional constraints:

- We do the PE# allocation *after* the 64-bit space has been assigned because the addresses we use directly determine the PE#. We then update the M32 PE# for the devices that use both 32-bit and 64-bit spaces or assign the remaining PE# to 32-bit only devices.
- We cannot "group" segments in HW, so if a device ends up using more than one segment, we end up with more than one PE#. There is a HW mechanism to make the freeze state cascade to "companion" PEs but that only works for PCIe error messages (typically used so that if you freeze a switch, it freezes all its children). So we do it in SW. We lose a bit of effectiveness of EEH in that case, but that's the best we found. So when any of the PEs freezes, we freeze the other ones for that "domain". We thus introduce the concept of "master PE" which is the one used for DMA, MSIs, etc., and "secondary PEs" that are used for the remaining M64 segments.

We would like to investigate using additional M64 windows in "single PE" mode to overlay over specific BARs to work around some of that, for example for devices with very large BARs, e.g., GPUs. It would make sense, but we haven't done it yet.

3. Considerations for SR-IOV on PowerKVM

- SR-IOV Background

  The PCIe SR-IOV feature allows a single Physical Function (PF) to support several Virtual Functions (VFs). Registers in the PF's SR-IOV Capability control the number of VFs and whether they are enabled.

  When VFs are enabled, they appear in Configuration Space like normal PCI devices, but the BARs in VF config space headers are unusual. For a non-VF device, software uses BARs in the config space header to discover the BAR sizes and assign addresses for them. For VF devices, software uses VF BAR registers in the *PF* SR-IOV Capability to discover sizes and assign addresses. The BARs in the VF's config space header are read-only zeros.

  When a VF BAR in the PF SR-IOV Capability is programmed, it sets the base address for all the corresponding VF(n) BARs. For example, if the PF SR-IOV Capability is programmed to enable eight VFs, and it has a 1MB VF BAR0, the address in that VF BAR sets the base of an 8MB region. This region is divided into eight contiguous 1MB regions, each of which is a BAR0 for one of the VFs. Note that even though the VF BAR describes an 8MB region, the alignment requirement is for a single VF, i.e., 1MB in this example.

There are several strategies for isolating VFs in PEs:

- M32 window: There's one M32 window, and it is split into 256 equally-sized segments. The finest granularity possible is a 256MB window with 1MB segments. VF BARs that are 1MB or larger could be mapped to separate PEs in this window. Each segment can be individually mapped to a PE via the lookup table, so this is quite flexible, but it works best when all the VF BARs are the same size. If they are different sizes, the entire window has to be small enough that the segment size matches the smallest VF BAR, which means larger VF BARs span several segments.
- Non-segmented M64 window: A non-segmented M64 window is mapped entirely to a single PE, so it could only isolate one VF.
- Single segmented M64 windows: A segmented M64 window could be used just like the M32 window, but the segments can't be individually mapped to PEs (the segment number is the PE#), so there isn't as much flexibility. A VF with multiple BARs would have to be in a "domain" of multiple PEs, which is not as well isolated as a single PE.
- Multiple segmented M64 windows: As usual, each window is split into 256 equally-sized segments, and the segment number is the PE#. But if we use several M64 windows, they can be set to different base addresses and different segment sizes. If we have VFs that each have a 1MB BAR and a 32MB BAR, we could use one M64 window to

assign 1MB segments and another M64 window to assign 32MB segments.

Finally, the plan to use M64 windows for SR-IOV, which will be described more in the next two sections. For a given VF BAR, we need to effectively reserve the entire 256 segments (256 * VF BAR size) and position the VF BAR to start at the beginning of a free range of segments/PEs inside that M64 window.

The goal is of course to be able to give a separate PE for each VF.

The IODA2 platform has 16 M64 windows, which are used to map MMIO range to PE#. Each M64 window defines one MMIO range and this range is divided into 256 segments, with each segment corresponding to one PE.

We decide to leverage this M64 window to map VFs to individual PEs, since SR-IOV VF BARs are all the same size.

But doing so introduces another problem: total_VFs is usually smaller than the number of M64 window segments, so if we map one VF BAR directly to one M64 window, some part of the M64 window will map to another device's MMIO range.

IODA supports 256 PEs, so segmented windows contain 256 segments, so if total_VFs is less than 256, we have the situation in Figure 1.0, where segments [total_VFs, 255] of the M64 window may map to some MMIO range on other devices:

```
0      1                     total_VFs - 1
+------+------+-     -+------+------+
|      |      |  ...  |      |      |
+------+------+-     -+------+------+

                      VF(n) BAR space

0      1                     total_VFs - 1                255
+------+------+-     -+------+------+-      -+------+------+
|      |      |  ...  |      |      |   ...  |      |      |
+------+------+-     -+------+------+-      -+------+------+

                      M64 window

           Figure 1.0 Direct map VF(n) BAR space
```

Our current solution is to allocate 256 segments even if the VF(n) BAR space doesn't need that much, as shown in Figure 1.1:

```
0      1                     total_VFs - 1                255
+------+------+-     -+------+------+-      -+------+------+
|      |      |  ...  |      |      |   ...  |      |      |
+------+------+-     -+------+------+-      -+------+------+

                      VF(n) BAR space + extra

0      1                     total_VFs - 1                255
+------+------+-     -+------+------+-      -+------+------+
|      |      |  ...  |      |      |   ...  |      |      |
+------+------+-     -+------+------+-      -+------+------+

                      M64 window

           Figure 1.1 Map VF(n) BAR space + extra
```

Allocating the extra space ensures that the entire M64 window will be assigned to this one SR-IOV device and none of the space will be available for other devices. Note that this only expands the space reserved in software; there are still only total_VFs VFs, and they only respond to segments [0, total_VFs - 1]. There's nothing in hardware that responds to segments [total_VFs, 255].

4. Implications for the Generic PCI Code

The PCIe SR-IOV spec requires that the base of the VF(n) BAR space be aligned to the size of an individual VF BAR.

In IODA2, the MMIO address determines the PE#. If the address is in an M32 window, we can set the PE# by updating the table that translates segments to PE#s. Similarly, if the address is in an unsegmented M64 window, we can set the PE# for the window. But if it's in a segmented M64 window, the segment number is the PE#.

Therefore, the only way to control the PE# for a VF is to change the base of the VF(n) BAR space in the VF BAR. If the PCI core allocates the exact amount of space required for the VF(n) BAR space, the VF BAR value is fixed and cannot be changed.

On the other hand, if the PCI core allocates additional space, the VF BAR value can be changed as long as the entire VF(n) BAR space remains inside the space allocated by the core.

Ideally the segment size will be the same as an individual VF BAR size. Then each VF will be in its own PE. The VF BARs (and therefore the PE#s) are contiguous. If VF0 is in PE(x), then VF(n) is in PE(x+n). If we allocate 256 segments, there are (256 - numVFs) choices for the PE# of VF0.

If the segment size is smaller than the VF BAR size, it will take several segments to cover a VF BAR, and a VF will be in several PEs. This is possible, but the isolation isn't as good, and it reduces the number of PE# choices because instead of consuming only numVFs segments, the VF(n) BAR space will consume (numVFs * n) segments. That means there aren't as many available segments for adjusting base of the VF(n) BAR space.