DMA representations IOMMU, sg chaining, etc

FUJITA Tomonori

fujita.tomonori@lab.ntt.co.jp

NTT Cyber Space Laboratories

IOMMU issues

- Ignoring LLDs' restrictions
  - Segment length
  - Segment boundary
- DMA parameters duplicated in many structures
  - struct device, request_queue, and device_dma_parameters
- Performance
  - Space management algorithm
  - IOMMU API changes

Let’s ignoring LLDs’ restrictions

LLD's restrictions: too long segment length

- Some LLDs have restrictions on segement length
  - e.g. bnx2 can't handle more than 64KB
- We have two places to merge pages (leads to larger segment than page size)
  - The block layer respects q->max_segment_size
  - IOMMUs merges as many pages as they like with ignoring the restrictions
- Some LLDs have a workaround to split too large segments

LLD's restrictions: spanning segment boundary

- Some LLDs have restrictions on segement boundary
  - e.g. Some ATAs can't handle a segment spanning 64K boundary
- Again we have two places to create segments spanning the boundary
  - The block layer respects q->seg_boundary_mask when it merges pages
  - IOMMUs maps segments to whatever memory area they like (which cloud span the boundanry) to ruin the block layer's efforts
- Some LLDs have a workaround to split segments spanning the boundary

The issues to solve

- IOMMUs can't see the device restrictions
  - The restrictions are stored in request queue (IOMMU can't access to)
  - IOMMU can see only struct device
    - e.g. dma_map_single(struct device, addr, len, dir)
- All the IOMMUs need to be fixed to support the restrictions

New device_dma_parameters structure

```
struct device_dma_parameters {
    unsigned int max_segement_size;
    unsigned long segment_boundary_mask;
};

struct pci_dev {
    struct device_dma_parameters dma_parms;
    struct device;
    …;
};

struct device {
    struct dvice_dma_parameters *dma_parms;
    …;
};
```

- device_dma_parameters is embedded in pci_dev (it will be in other dma'able devices)
- struct device has a pointer to struct device_dma_parameters

What IOMMUs were fixed?

- Segment length
  - x86_64 (gart)
  - Alpha
  - POWER
  - PARISC (sba, ccio)
  - IA64
  - SPARC64

Blue: patch merged
green: patch submitted
Red: not yet

- Segment boundary
  - x86_64 (calgary, gart, Intel)
  - Alpha
  - POWER
  - PARISC (sba, ccio)
  - IA64
  - SPARC64
  - ARM (jazzdma.c)
  - swiotlb (x86_64, ia64)

Let’s store LLDs’ restrictions at three different locations

dma parameters are confusing

- struct device has
  - u64* dma_mask
  - u64 coherent_dma_mask
  - struct device_dma_parameters *dma_parms
- sturct device_dma_parameters has
  - unsigned int max_segment_size;
  - unsigned long segment_boundary_mask
- struct request_queue has
  - unsigned int max_segment_size
  - unsigned long seg_boundary_mask

Needs to clean up dma parameters

- Struct device are also used for non dma'able devices so should not have
  - u64* dma_mask
  - u64 coherent_dma_mask
- The block layer and IOMMUs duplicate the same values
  - Max_segment_size
  - Segment_boudnary_mask

IOMMU is becoming
the performance bottleneck

What's the best algorithm to mange free space?

- IOMMUs spend long time to mange free space
  - Most of use simple bitmap
  - Intel uses Red Black Trees
    - I converted POWER iommu to use it and lost 20% of performance with netperf.
  - What's the best (depends on the size of IOMMU memory space)
- Should we have one library functions for IOMMU
  - It's really hard since every IOMMUs use the own techniques
  - lib/iommu-helper.c provides primitive functions for bitmap management

When should we flush IOTLB?

- Flushing IOTLB is expensive
  - Most of IOMMUs delay flushing IOTLB entries until they are reused
  - Intel IOMMU (VT-d) flushes IOLTB entries every time the entries are unmapped
- How to avoid IOTLB flush
  - The drivers should batch unmapping?
  - Dividing IOMMU space and assigning them to each drivers?

When should we flush IOTLB?

- Flushing IOTLB is expensive
  - Most of IOMMUs delay flushing IOTLB entries until they are reused
  - Intel IOMMU (VT-d) flushes IOLTB entries every time the entries are unmapped
- How to avoid IOTLB flush
  - The drivers should batch unmapping?
  - Dividing IOMMU space and assigning them to each drivers?

Why should we unmap?

- Decent hardware handles 64 bit space
- Nice IOMMU also handles large space (64 bit)
- Just map all the host memory and don’t unmap at all
- We lose some features (like protection) but it would be nice in some circumstances

SCSI data accessors, SG chaining, SG ring, etc

What's scsi data accessors?

- Helper functions to insulate LLDs from data transfer information
  - We planed to make lots of changes to scsi_cmnd structure support sg chaining and bidirectional data transfer
  - LLDs directly accessed to the values in scsi_cmnd
  - We rewrited LLDs to access scsi_cmnd via new accessors

scsi data accessors example access to scsi_cmnd's sg list

Old way

```
struct scsi_cmnd *sc
struct scatterlist *sg =
        sc->request_buffer;
```

New way

```
struct scsi_cmnd *sc
struct scatterlist *sg =
        scsi_sglist(sc);
```

```
#define scsi_sglist(sc)
sc->request_buffer
```

struct scsi_cmnd changed

2.6.24

```
struct scsi_cmnd {
   void *request_buffer;
```

Post 2.6.24

```
struct sg_table {
    struct scatterlist *sgl;

struct scsi_data_buffer {
    struct sg_table table;

struct scsi_cmnd {
    struct scsi_data_buffer sdb;
```

We just changed scsi_sglist macro, not all the drivers

```
#define scsi_sglist(sc)
sc->request_buffer
```

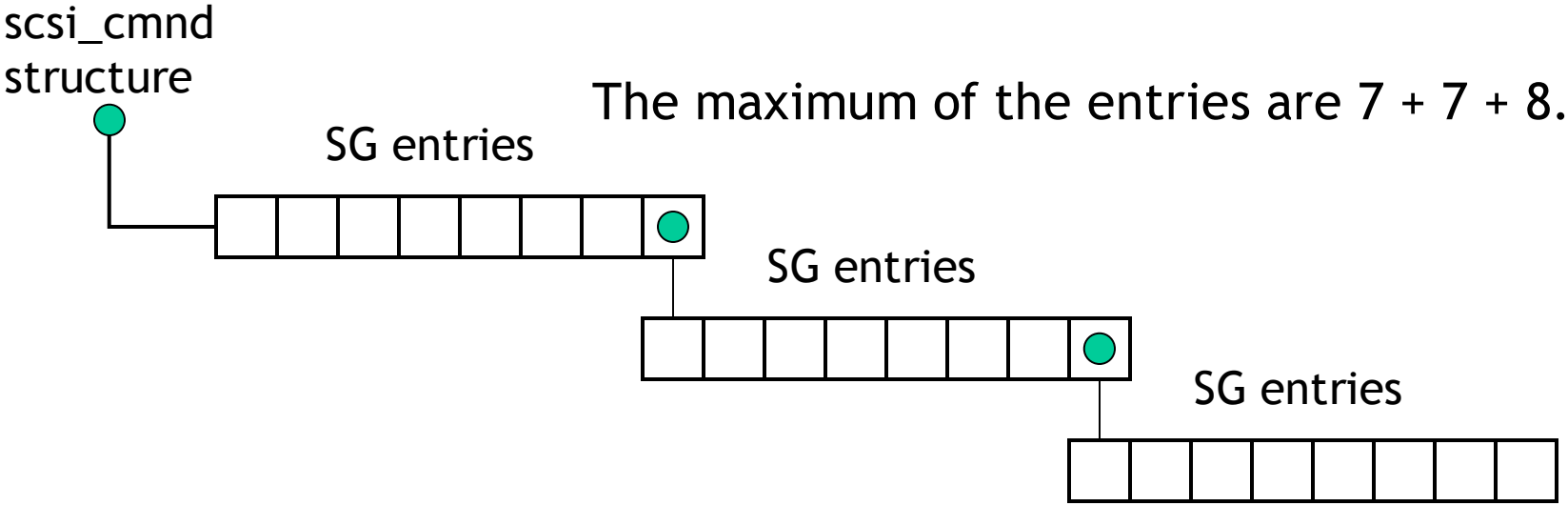
```
#define scsi_sglist(sc)
sc->sdb.table.sgl
```

scatter gather chaining

- SCSI-ml couldn’t handle Large data transfer
  - scsi-ml pools 8, 16, 32, 64, and 128 sg entries (the sg size is 32 bytes on x86_64)
  - People complains about scsi memory consumption so we can’t have large sg entries
  - scsi_cmnd struct has a point to sg entries

scatter gather chaining (cont.)

- sg chaining
  - The last sg entry tells us it's the last entry or we have more sg entries
    - The last sg entry points to the first entry of the next sg list
  - sg entries aren't continuous any more!



scsi data accessors (cont.)
Too simple sg setup examples

How a LLD tell addresses for I/Os for the HBA

Old way

```
struct scsi_cmnd *sc
struct scatterlist *sg =
          sc->request_buffer;

for(i = 0; i < nseg; i++) {
   paddr = sg_dma_address(sg[i]);
   …
```

New way

```
stcuct scsi_cmnd *sc
struct scatterlist *sg;

scsi_for_each_sg(sc, sg, nseg,i) {
   physaddr = sg_dma_address(sg);
   …
```

How didi scsi data accessors help sg chaining?

- Before sg chaining

```
#define scsi_for_each_sg(sc, sg, nseg, i)
for(i = 0, sg = scsi_sglist(sc); i < nseg, i++, sg++)
```

sg entries must be continuous

- We changed it after sg chaining

```
#define scsi_for_each_sg(sc, sg, nseg, i)
for(i = 0, sg = scsi_sglist(sc); i < nseg, i++, sg =
   sg_next(sg))
```

sg_next macro takes care of discontinuous sg entries

LLDs can support sg chaining magically without modifications

SG chaining isn't good?

- Some wants something like sg chaing
  - Crypto already has something, virto wanted it
- Difficult to modify SG chaining once creating it
  - Can't add new entries to it or split it easily
- SCSI (and block) drivers shouldn't manipulate SG lists
  - Building sg lists is the job for the block and scsi mid-layer
  - The drain buffer work and the IOMMU fixes enables us to remove SG modifying code in libata

SG ring: two level traversal

- Struct sg_ring has a list_head and a scatter list
- We chain sg_ring structures with the list_head
- SCSI tried a similar idea (scsi_sgtable) before

```
struct sg_ring {
   struct list_head list;
   int num, max;
   struct scatterlist sg[0]
};
```

SG table:

- It has just a sg list and the number of the sg entries.
- We chain the sg list as SG chain

```
struct sg_table {
   struct scatterlist *sg;
   unsigned int nents;
   unsigned int orig_nents;
};
```