# Page migration for IOMMU enhanced devices

Tomasz Stanislawski

Samsung R&D Institute Poland

August 20, 2013

# Agenda

# IOMMU

- IOMMU allows to create a flexible mapping of device-visible virtual addresses to physical addresses.
- Kernel provides lowlevel framework for controlling mapping space by using a concept of IOMMU domain. Refer to drivers/iommu.c
- if possible use high order pages due to limited capacity of TLB
- configuration of IOMMUs is hidden below DMA API for ARM platforms

# DMA API - dma_alloc_coherent

DMA API is used to manage buffers for DMA operations.
The dma_alloc_coherent() is used to allocate memory. Allocation steps for IOMMU enhanced devices and ARM platforms are following:

1. allocate pages of orders from 0 to 4 from BUDDY
2. create DMA mapping
    - reserve a contiguous chunk of DMA address space
    - map a chunk to IOMMU using iommu_map()
3. create a kernel mapping in VMALLOC area (optional)

# Page Migration

Pages can be divided into two categories

- MOVABLE - can be migrated
  - page cache
  - anonymous pages
  - KSM pages (Kernel Shared Memory)
- NONMOVABLE - cannot be migrated
  - all others, including memory for DMA

# Page Migration

The mechanism of page migration allows to change physical location of pages.

- Initially dedicated for NUMA-based architectures to bring pages closer to CPU that runs a process that uses those pages
- Used by memory compaction to reduce external fragmentation
- Used by Contiguous Memory Allocator (CMA) to clean contiguous chunks of memory for DMA allocation

# Problem

Problems with IOMMU pages allocated by DMA API

- allocation of 4-order page is very likely to fail
- pages are non-movable therefore they cannot use CMA area
- non-movable low-order pages cause external fragmentation

# Solution

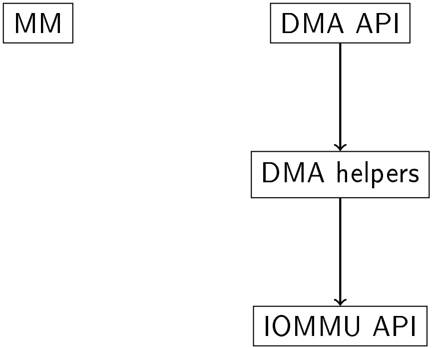
Add support for migration for pages mapped by IOMMUs.
Issues:

- DMA API is a low-level API, migration is a relatively high level feature
- migration during DMA operation
- migration during user access via mmap() mappings

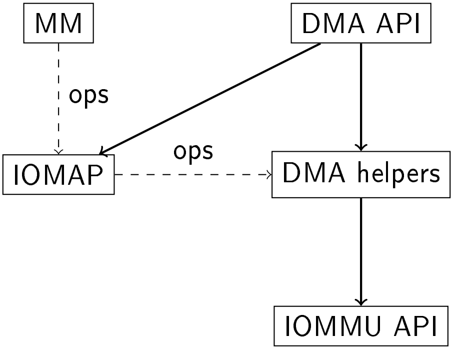
# MM mapping

- Every mapping holds a radix tree to track which parts of a file are already in RAM and where.
- The VFS subsystem supports page migration by dedicated ops in *structaddress_space* AKA mapping.
- Prior to migration, all user mappings are removed and every process trying to access migrated data is put to sleep

# Diagram



# Diagram



# MM extensions

```
int migrate_page(struct address_space *mapping,
                 struct page *newpage, struct page *page,
                 enum migrate_mode mode)
{
        int rc;

        rc = migrate_page_move_mapping(mapping, newpage, page, NULL, mode);
        if (rc)
                return rc;

        migrate_page_copy(newpage, page);
        return 0;
}
```

- called after all user's page mapping are removed
- all processes accessing a page are put to sleep
- migrate_page_move_mapping() replaces page in radix tree
- migrate_page_copy copies() page content and flags

# MM extensions

A closer look on *migrate_page_copy*.

```
void migrate_page_copy(...)
{
        if (PageHuge(page))
                copy_huge_page(newpage, page);
        else
                copy_highpage(newpage, page);

        ... copy page flags
}
```

HW may modify page content during page copying.

# MM extensions

Add migratepagecopy to struct address_space_operations.

- overloading standard mechanism for page locking
- use additional methods for cache synchronization
- provide additional methods for data synchronization

# MM extensions

```
void migrate_page_copy(...)
{

        if (mapping && mapping->a_ops->migratepagecopy)
                mapping->a_ops->migratepagecopy(
                        mapping, newpage, page);

        else if (PageHuge(page))
                copy_huge_page(newpage, page);
        else
                copy_highpage(newpage, page);
        ... copy page flags
}
```

# IOMAP

IOMAP is a new subsystem between DMA and MM mappings.

- provide a special type of dummy address_space that supports migration but no read/write ops.
- allow DMA layer to insert a page at a given offset i.e. 10th page of a buffer
- use radix tree to track physical location of buffer pages
- inform DMA layer about an event of a page being migrated
- support mmap()
- mark DMA page as private because it is referenced by DMA
- relatively short and simple code due to reusing MM helpers

# IOMAP - API

```
struct iomap {
        struct address_space mapping;
        const struct iomap_client_ops *cops;
};

int iomap_create(struct iomap *iomap,
        const struct iomap_client_ops *cops);

void iomap_release(struct iomap *iomap);

int iomap_insert_page(struct iomap *iomap,
        struct page *page, unsigned index);

void iomap_remove_page(struct iomap *iomap,
        struct page *page);

int iomap_mmap(struct iomap *iomap,
        struct vm_area_struct *vma,
        pgprot_t prot);

int PageIomap(struct page *page);
```

# IOMAP - create/release

```
int iomap_create(struct iomap *iomap,
        const struct iomap_client_ops *cops)
{
        address_space_init_once(&iomap->mapping);
        iomap->mapping.a_ops = &iomap_aops;
        iomap->mapping.backing_dev_info = &noop_backing_dev_info;
        iomap->cops = cops;

        return 0;
}

void iomap_release(struct iomap *iomap)
{
}
```

# IOMAP - insert/remove

```
int iomap_insert_page(struct iomap *iomap,
        struct page *page, unsigned index)
{
        int ret;
        SetPagePrivate(page);
        ret = add_to_page_cache_lru(page,
                &iomap->mapping, index, GFP_KERNEL);
        if (ret) {
                ClearPagePrivate(page);
                return ret;
        }
        unlock_page(page);
        return 0;
}

void iomap_remove_page(struct iomap *iomap,
        struct page *page)
{
        lock_page(page);
        /* page might have been migrated prior to lock_page */
        if (page_mapping(page))
                delete_from_page_cache(page);
        ClearPagePrivate(page);
        unlock_page(page);
}
```

# IOMAP - mmap()

Pseudocode:

```
int iomap_mmap(struct iomap *iomap,
        struct vm_area_struct *vma,
        pgprot_t prot)
{
        for each index:
                rcu_read_lock();
                page = radix_tree_lookup(&mapping->page_tree, index);
                vm_insert_page(vma, uaddr, page);
                rcu_read_unlock();
                uaddr += PAGE_SIZE;
}
```

# IOMAP - DMA iterations

```
void iomap_migratepagecopy(struct address_space *mapping,
        struct page *newpage, struct page *page)
{
        struct iomap *iomap = to_iomap(mapping);

        iomap->cops->migratepagecopy(iomap,
                newpage, page);
}

struct address_space_operations iomap_aops = {
        .migratepagecopy = iomap_migratepagecopy,
        .migratepage = migrate_page,
        ...
};
```

# IOMAP - other

By default, MM expects that every private page is associated with buffer cache. Moreover, MM assumes that page private field is a buffer head leading to segmentation faults. This is prevented by overloading some ops with dummy functions.

```
static int iomap_set_page_dirty(struct page *page)
{
        return 0;
}

static int iomap_releasepage(struct page *page, gfp_t gft)
{
        return 0;
}
struct address_space_operations iomap_aops = {
        ...
        .set_page_dirty = iomap_set_page_dirty,
        .releasepage = iomap_releasepage,
};
```

# IOMMU extension

Add a support for migration of given DMA space to a new physical address.

```
void iommu_migrate(struct iommu_domain *domain,
        unsigned long iova,
        phys_addr_t paddr, size_t size,
        void (copy_cb)(void *), void *cb_priv);
```

| | |
|---|---|
| iova | DMA address |
| paddr | new physical address |
| size | size of mapping |
| copy_cb | function used to copy data content |
| cb_priv | extra parameter for copy_cb |

# IOMMU extension

The iommu_migrate executes ops from domain driver that:

1. block IOMMU - to protect from IOMMU faults
2. unmap previous mapping at *iova* to *iova* + *size*
3. call *copy_cb* to copy data content
4. map *paddr* to *iova* to *iova* + *size*
5. unblock IOMMU

# DMA API modifications for ARM

The DMA buffer is described by an array of pages. This array is kept in:

- a range of pages in *atomic_pool* if *cpu_addr* lays in the area for atomic mappings
- directly under *cpu_addr* if a buffer is allocated with *DMA_ATTR_NO_KERNEL_MAPPING* attribute
- in *vm_struct* :: *pages* associated with an area in *VMALLOC* where *cpu_addr* lies

# DMA API modifications for ARM

Calls to DMA API from IOMAP needs to call *iommu_migrate*(). Therefore *iomap* has to be converted to *pages* and *dma_addr* and proper *dev*. Achieved by embedding *pages* into a bigger structure.

```
struct page **pages;
```

→

```
struct __iommu_buffer {
    struct iomap iomap;
    struct device *dev;
    dma_addr_t dma_addr;
    struct page *pages[];
};
```

# DMA API modifications for ARM

Updated dma_alloc_coherent() for IOMMU devices

- allocate *struct __iommu_buffer*
- initialize IOMAP with *__iommu_migratepagecopy* ops
- allocate pages of order 0 from BUDDY
- create DMA mapping
  - allocate a contiguous chunk of DMA address space
  - map a chunk to IOMMU using iommu_map()
- map to VMALLOC area (optional)
- insert all pages to IOMAP using *iomap_insert_page*()

# DMA API modifications for ARM

DMA implements an *migratepagecopy* ops called by IOMAP.

```
static void __iommu_migratepagecopy(struct iomap *iomap,
        struct page *to, struct page *from)
{
        struct page *pages[2] = { to, from };
        struct __iommu_buffer *ib = to_iobuf(iomap);
        dma_addr_t dma_addr = ib->dma_addr +
                (to->index << PAGE_SHIFT);

        iommu_migrate(domain, dma_addr,
                page_to_phys(to), PAGE_SIZE,
                __iommu_cb, pages);
        get_page(to);
        ib->pages[to->index] = to;
        put_page(from);
}
```

# DMA API modifications for ARM

Copy-callback implemented by DMA layer called from IOMMU layer.

```
static void __iommu_cb(void *cookie)
{
        struct page **pages = cookie;
        struct page *to = pages[0];
        struct page *from = pages[1];

        /* synchronize source page to CPU */
        __dma_page_dev_to_cpu(from, 0, PAGE_SIZE, DMA_BIDIRECTIONAL);
        copy_highpage(to, from);
        /* assure that destination is flushed to RAM */
        __dma_page_cpu_to_dev(to, 0, PAGE_SIZE, DMA_BIDIRECTIONAL);
}
```

# Summary

The process of migration of IOMMU pages is done in steps:

1. page migration in MM part:
   - remove all mappings, block user processes
   - call *migratepage* chained to *migrate_page*() by IOMAP
   - remove a page from a radix tree
   - call *migratepagecopy*
2. IOMAP part: chain to DMA layer by *migratepagecopy* ops

# Summary

1. DMA part: call *iommu_migrate*()
2. IOMMU part:
   1. block IOMMU - to protect from IOMMU faults
   2. unmap previous mapping at *iova* to $iova + size$
   3. call *copy_cb* to copy data content, back to DMA layer
      - invalidate CPU cache for source
      - copy data
      - flush CPU cache to RAM
   4. map *paddr* to *iova* to $iova + size$
   5. unblock IOMMU
3. DMA part II: update *pages* array

- fix layering between IOMAP and DMA and IOMMU
- migrate pages without blocking whole IOMMU domain
- how to deal with kernel mappings? Only allocations with *DMA_ATTR_NO_KERNEL_MAPPING* works.
- allow only 0-order pages from DMA to be migrated
- associate IOMAP object with IOMMU domain rather than DMA buffer
- support migration for pages in shared buffers

# Fix layering

Add support for block/unblock calls to IOMMU API. Modify IOMAP to migrate pages only when IOMMU domain is blocked.

```
int iomap_migratepage(struct page *newpage,
        struct page *page, ...)
{
        int ret;

        iommu_block();
        ret = migrate_page(newpage, page)
iomm_migrate(newpage, page);
        iommu_unblock();

        return ret;
}
```

# Non-blocking migration

Use fault handling in IOMMU to detect if page was accessed.
Migration pseudocode:

1. mark page as non-dirty
2. mark an entry in IOMMU's page table as a fault-on-write entry
3. invalidate CPU cache for source
4. copy data
5. flush CPU cache to RAM
6. if a page is dirty goto 1

# Non-blocking migration

IOMMU fault pseudocode:

1. restore a page-table entry to a state before migration
2. mark a page as dirty

# Questions?