

The DRM video memory manager

Contents

1 Purpose and scope	3
1.1 Definitions	3
2 How we solve the problems	4
2.1 Memory resources are available to all clients	4
2.2 We should not run out of available memory	4
2.3 Only a single copy of a memory buffer should be maintained	5
2.4 A memory buffer should not unknowingly disappear when we want to work with it or when the GPU is working with it.	5
2.4.1 GPU	5
2.4.2 CPU	5
2.5 Reading from video memory shouldn't be a dead slow operation	6
2.6 The memory manager should be fast. Should it really live in the kernel?	6
2.6.1 Sub-allocators	6
2.7 Buffers should be shareable	7
2.8 What about pinned buffers	7
3 Current status	7
4 Typical usage pattern	8
4.1 Accelerated 3D	8
4.2 Accelerated video decoding (XvMC)	9
4.2.1 Decoding thread	9

1 Purpose and scope

With the memory manager implementation, we basically try to solve the following problems.

- Memory resources are available to all clients.
- We should not run out of available video memory.
- Only a single copy of a memory buffer should be maintained.
- A memory buffer should not unknowingly disappear when we want to work with it or when the GPU is working with it.
- Reading from video memory should not be a dead slow operation.
- The memory manager should be fast. Should it really live in the kernel?
- Buffers should be sharable, for example X server redirected drawables to DRI clients.

1.1 Definitions

We need to make some definitions, and we start by defining two basic types of memory:

- **Fixed memory** memory, which appear in a fixed PCI location to the memory manager. This memory is typically on-card VRAM or pre-bound AGP memory. For example, the X server can bind 128M of memory to the AGP aperture and tell the memory manager that this is a region of fixed memory.
- **Translation-Table memory** A set of system memory pages that can be accessed by the GPU by dynamically binding them to a translation-table, for example the AGP aperture.

Both these types of memory can be either cache coherent or non-cache coherent. Legacy hardware in general only support non-cache-coherent memory which has the following drawbacks:

- Reading from it is very slow since processors generally don't prefetch this memory. 10MiB/s is a typical figure.
- Transition from cache-coherent to non-cache-coherent memory has a long latency associated with it, since a global CPU cache- and TLB flush needs to take place. This affects the performance of translation-table memory, but can probably be optimized somewhat in the future. Typically this latency is equivalent to reading a 4KiB page of non-cache-coherent memory or writing about 256KiB of write-combined AGP memory. The latency is, however, largely independent of the number of pages transitioned from cache-coherent to non-cache-coherent and vice versa in a single operation.

Also, as pointed out recently on the dri-devel list, Fixed memory can be mappable by the CPU or not mappable.

2 How we solve the problems

2.1 Memory resources are available to all clients

We can make sure that this is the case by managing memory in the kernel. The kernel is aware of all memory buffers and can evict them to either other memory types or secondary storage as needed. The X server can set up a maximum of 7 memory managers for different memory regions of fixed and translation-table memory. A typical implementation would have one manager for VRAM, one manager for pre-bound AGP memory and one manager for translation-table AGP memory. The client decides in which memory types a buffer may live, and the device specific DRM driver decides where it goes when it is evicted. VRAM, for example, may be evicted to AGP if there is a fast VRAM to AGP copy operation, or to system translation-table memory if there is support for fast PCI scatter-gather DMA.

2.2 We should not run out of available memory

Of the listed memory types, translation-table memory is the most abundant. If we need more translation-table memory, we just allocate a couple of more pages from system RAM, call them a buffer and bind them to the translation-table. If we run out of translation-table (read AGP aperture) space, we start evicting other translation-table buffers, and even the buffers of other clients. Unbinding is a fast operation *as long as we don't change the caching policy*.

If we desperately need more VRAM space, we can evict unused VRAM buffers to translation-table memory (provided there is a fast blit- or DMA operation). Again translation-table memory is abundant.

translation-table memory requires that pages are locked in the kernel, and we cannot unfortunately lock too many pages. At some point we would render the system unusable, and any DRI user app can do it. So we need to set a limit on how many pages we can lock. When such a limit is exceeded we would like to release pages that we haven't used for a while to the standard swapping system. Unfortunately there is no easy way to do that in the kernel. Swappable pages need to belong to a user space process and should have no other references that the kernel memory subsystem knows about. The way the kernel people suggest us to do this is to have the X server `mmap()` a **big** anonymous area that it tells the video memory manager about. The Linux `mmap()` operation is lazy and doesn't allocate any pages, so this is not a big waste of memory. When we need to unlock buffer pages that haven't been used for a while we simply populate the X server map with them, remember where we put them and then we unlock them. The kernel will then start to swap them out on secondary storage when needed, and we can easily get them back by telling the kernel we need an X server page at user address `xxx`.

Currently we allow half the amount of physical memory below 1GB (4GB on 64-bit machines) to be locked as translation-table memory.

2.3 Only a single copy of a memory buffer should be maintained

We can do this by always physically copying buffer contents that needs to be evicted, since we assume that we have a fast copy operation. translation-table memory doesn't even need this since evicting just unbinds it from the translation table.

2.4 A memory buffer should not unknowingly disappear when we want to work with it or when the GPU is working with it.

2.4.1 GPU

To make sure a buffer is not evicted while the GPU is working with it, before we send any GPU commands that reference the buffer we *validate* it. This makes sure the buffer is placed where we want it to be and that it is not evicted. Then we associate it with a *fence* object. The fence object has a place in the GPU command stream and expires or *signals* when the GPU is done with all commands preceding it in the command stream. After a buffer's fence object has signaled, it is safe to evict the buffer if needed. Many buffers can be associated with the same fence object and fence objects can also be accessed from user space. A driver can even implement various types of fence objects depending on the content of the buffer (video / 3D) and the potential flush operations that need to be completed by the hardware.

The simplest form of a fence object implementation is to just wait for GPU idle when we wait for a fence object to expire. A slightly more advanced form is to add a blit operation of a sequence number to the command stream and wait for that number to be blitted.

The i915 DRM driver has a more elaborate implementation with command stream user IRQs, and different fence types for command buffers and buffers that need a read / write flush operation before they can be evicted.

2.4.2 CPU

How do we make sure that a buffer is not evicted while the CPU is writing to it, and how can we find it from the driver when it has been evicted? The simple answer is that we don't have to care. Just go ahead and write to it or read from the buffer.

This is one of the key features of the memory manager. It allows a certain class of performance optimizations (sub-allocators described in section 2.6.1) and generally makes driver writing easier.

This trick is done using CPU page table manipulation. Before a buffer is evicted or otherwise moved, all user space page tables referencing it are killed. When the CPU tries to access a page in the buffer we simply update the page table entry to where the buffer is currently located. If it is on secondary memory we read it in. If it is in unmappable VRAM, we copy it to mappable VRAM with the blitter. The process trying to read or write from the buffer will not be aware of this, but it may be stalled for a little while if the buffer needs to be moved to be mappable.

The current implementation (translation-table memory only) works like this: When a buffer needs to be evicted, its user space page tables are killed. It is then unbound from AGP. If it is then validated, it is simply rebound to AGP. If it is accessed by the CPU while *not* bound to AGP it is

first transitioned from non-cache-coherent to cache-coherent, and then the user page table entry is updated to point to it. The transition is slow but does typically not occur very often.

2.5 Reading from video memory shouldn't be a dead slow operation

Due to the fact that processors don't prefetch non-cache-coherent memory, reading from it is very slow. Typically around 10MB/s. There are different ways to speed this up:

- Move translation-table memory out of the aperture and make it cache coherent. This operation has a latency typically to reading one-to two pages so the region read should be larger for this to have a performance impact.
- Hardware assisted copy to cache-coherent memory. This can, for example, be a PCI DMA scatter-gather operation or a hardware blit to cache-coherent translation-table memory (if available).

For large buffers, the first method is probably the most attractive one but for small amounts of data, the second method is superior. The DRI driver can implement a suitable policy from user space.

2.6 The memory manager should be fast. Should it really live in the kernel?

The current i915tex driver is a proof of concept that shows that, in general, things work quite well. It currently implements only non-cache-coherent translation-table memory, and thus suffers from the annoying caching policy changing latency when buffers are created and destroyed. The overhead of the increased number of kernel IOCTLs seems to be very small.

2.6.1 Sub-allocators

When porting the i915tex driver, it became apparent that when creating and destroying batch-buffers, the caching policy changing latency became too large and had a severe impact on performance. Therefore a batch buffer pool was created using a large memory manager buffer which was in turn managed in user space using a sub-allocator for fixed size batch-buffers.

This can probably also be the way to get the most performance out of texture memory implementations. Memory would be allocated from DRM in big chunks, which in turn are managed from user space. The DRM memory manager page table manipulation makes this possible, as the DRI drivers will never notice and need not care when / if a chunk is evicted by the kernel.

Sub-allocators can also be beneficial from a memory usage point of view. Because of the page-table manipulation, Kernel DRM buffer size needs to be page-size (4KiB) aligned. If a client uses many very small textures, sub-allocators can be a way of saving a substantial amount of memory.

2.7 Buffers should be shareable

All DRM buffers carry with them a unique identifier, and when they are created, the creating process indicates whether they should be shareable or not. Typical examples of shareable buffers are common back- and depth- buffers or redirected drawables.

When a client wants to share a buffer with another client it sends the buffer identifier to the other client(s), and the other clients call a libdrm function to increase the refcount of the buffer and obtain information about it.

Buffers are destroyed only when all references to them are gone.

2.8 What about pinned buffers

Buffers can carry a `NO_EVICT` and a `NO_MOVE` status. Buffers with `NO_EVICT` status are available only to the X server and can never be evicted. This flag is intended for scanout buffers. When the memory manager is cleaned (VT switching for example), `NO_EVICT` buffers are not allowed and will cause an error.

Buffers with `NO_MOVE` status are guaranteed to always show up in the same physical location to the GPU when they are validated. They can be evicted, but in principle they are only evicted when the memory manager is cleaned.

`NO_MOVE` buffers are considered bad, because they may cause memory manager fragmentation, and it's not clear that they are really needed. Perhaps there is a use for them for time-critical applications like XvMC video playback, where moving an evicted buffer back in may delay playback long enough to cause video jerkiness.

3 Current status

The current status (March 2007) is that translation-table memory is implemented and functional. The i915 DRM driver supports both cache-coherent and non-cache-coherent translation-table memory.

Support for fixed memory is implemented, but only tested with a pre-bound AGP region using a modified i915tex driver.

Accelerated CPU reading from buffers is not yet implemented in the i915tex driver, but since the i915 DRM driver supports cache-coherent translation-table memory this should be an easy task.

A user-space generic sub-allocator implementation is partly available in the Mesa dri/common directory. Currently is only used for i915tex batch buffers.

Secondary storage swapping is not implemented at this point.

4 Typical usage pattern

4.1 Accelerated 3D

The following is an example of a typical 3D usage pattern:

```
begin;
  Allocate_command_buffer;
  Allocate_texture_buffers;

  do{
    Map_buffers;
    Update_textures;
    Create_command_stream;
    Unmap_buffers;

    Lock_hardware;

    Validate_buffers;

    Map_command_buffer;
    Update_command_stream_with_new_buffer_offsets;
    Unmap_command_buffer;
    Submit_command_buffer;

    Fence_buffers;

    Unlock_hardware;

  }while(!Finished_drawing);

end;
```

4.2 Accelerated video decoding (XvMC)

4.2.1 Decoding thread

```
begin;
  Allocate_command_buffer;
  Allocate_frame_buffer_queue;

  do{
    Get_decoding_frame_buffer;
    Add_reference_if_becoming_reference_frame;

    Select_backward_reference_frame;
    Select_forward_reference_frame;

    Map_command_buffer;
    Create_command_stream;
    Unmap_command_buffer;

    Lock_hardware;
    Validate_buffers;

    Map_command_buffer;
    Update_command_stream_with_new_buffer_offsets;
    Unmap_command_buffer;
    Submit_command_buffer;

    Fence_buffers;

    Unlock_hardware;
    Release_unneeded_reference_frames;
  }while(!Finished_decoding);

end;
```

Here, the decoded frame is synchronized, blended, displayed and released in the displaying thread (not shown).

Note that in the examples above, mapping is really only a fast synchronization operation. Due to the previously mentioned page table tricks, buffers are available for reading and writing from the first map operation until they are destroyed.