Notes on the Generic Block Layer Rewrite in Linux 2.5

Notes Written on Jan 15, 2002:
Jens Axboe <jens.axboe@oracle.com>
Suparna Bhattacharya <suparna@in.ibm.com>

Last Updated May 2, 2002
September 2003: Updated I/O Scheduler portions
Nick Piggin <piggin@cyberone.com.au>

Introduction:

These are some notes describing some aspects of the 2.5 block layer in the context of the bio rewrite. The idea is to bring out some of the key changes and a glimpse of the rationale behind those changes.

Please mail corrections & suggestions to suparna@in.ibm.com.

Credits:

2.5 bio rewrite:
Jens Axboe <jens.axboe@oracle.com>

Many aspects of the generic block layer redesign were driven by and evolved over discussions, prior patches and the collective experience of several people. See sections 8 and 9 for a list of some related references.

The following people helped with review comments and inputs for this document:
Christoph Hellwig <hch@infradead.org>
Arjan van de Ven <arjanv@redhat.com>
Randy Dunlap <rdunlap@xenotime.net>
Andre Hedrick <andre@linux-ide.org>

The following people helped with fixes/contributions to the bio patches while it was still work-in-progress:
David S. Miller <davem@redhat.com>

Description of Contents:

Bio Notes

Let us discuss the changes in the context of how some overall goals for the block layer are addressed.

1. Scope for tuning the generic logic to satisfy various requirements

The block layer design supports adaptable abstractions to handle common processing with the ability to tune the logic to an appropriate extent depending on the nature of the device and the requirements of the caller. One of the objectives of the rewrite was to increase the degree of tunability and to enable higher level code to utilize underlying device/driver capabilities to the maximum extent for better i/o performance. This is important especially in the light of ever improving hardware capabilities and application/middleware software designed to take advantage of these capabilities.

1.1 Tuning based on low level device / driver capabilities

Sophisticated devices with large built-in caches, intelligent i/o scheduling optimizations, high memory DMA support, etc may find some of the generic processing an overhead, while for less capable devices the generic functionality is essential for performance or correctness reasons. Knowledge of some of the capabilities or parameters of the device should be used at the generic block layer to take the right decisions on behalf of the driver.

How is this achieved ?

Tuning at a per-queue level:

i. Per-queue limits/values exported to the generic layer by the driver

Various parameters that the generic i/o scheduler logic uses are set at a per-queue level (e.g maximum request size, maximum number of segments in a scatter-gather list, hardsect size)

Some parameters that were earlier available as global arrays indexed by major/minor are now directly associated with the queue. Some of these may move into the block device structure in the future. Some characteristics have been incorporated into a queue flags field rather than separate fields in themselves. There are blk_queue_xxx functions to set the parameters, rather than update the fields directly

Some new queue property settings:

blk_queue_bounce_limit(q, u64 dma_address)
Enable I/O to highmem pages, dma_address being the limit. No highmem default.

blk_queue_max_sectors(q, max_sectors)
Sets two variables that limit the size of the request.

- The request queue's max_sectors, which is a soft size in units of 512 byte sectors, and could be dynamically varied by the core kernel.

- The request queue's max_hw_sectors, which is a hard limit and reflects the maximum size request a driver can handle in units of 512 byte sectors.

The default for both max_sectors and max_hw_sectors is 255. The upper limit of max_sectors is 1024.

blk_queue_max_phys_segments(q, max_segments)
Maximum physical segments you can handle in a request. 128 default (driver limit). (See 3.2.2)

blk_queue_max_hw_segments(q, max_segments)
Maximum dma segments the hardware can handle in a request. 128 default (host adapter limit, after dma remapping). (See 3.2.2)

blk_queue_max_segment_size(q, max_seg_size)
Maximum size of a clustered segment, 64kB default.

blk_queue_hardsect_size(q, hardsect_size)
Lowest possible sector size that the hardware can operate on, 512 bytes default.

New queue flags:

QUEUE_FLAG_CLUSTER (see 3.2.2)
QUEUE_FLAG_QUEUED (see 3.2.4)

ii. High-mem i/o capabilities are now considered the default

The generic bounce buffer logic, present in 2.4, where the block layer would by default copyin/out i/o requests on high-memory buffers to low-memory buffers assuming that the driver wouldn't be able to handle it directly, has been changed in 2.5. The bounce logic is now applied only for memory ranges for which the device cannot handle i/o. A driver can specify this by setting the queue bounce limit for the request queue for the device (blk_queue_bounce_limit()). This avoids the inefficiencies of the copyin/out where a device is capable of handling high memory i/o.

In order to enable high-memory i/o where the device is capable of supporting it, the pci dma mapping routines and associated data structures have now been modified to accomplish a direct page -> bus translation, without requiring a virtual address mapping (unlike the earlier scheme of virtual address -> bus translation). So this works uniformly for high-memory pages (which do not have a corresponding kernel virtual address space mapping) and low-memory pages.

Note: Please refer to Documentation/PCI/PCI-DMA-mapping.txt for a discussion on PCI high mem DMA aspects and mapping of scatter gather lists, and support for 64 bit PCI.

Special handling is required only for cases where i/o needs to happen on pages at physical memory addresses beyond what the device can support. In these cases, a bounce bio representing a buffer from the supported memory range is used for performing the i/o with copyin/copyout as needed depending on the type of the operation. For example, in case of a read operation, the data read has to be copied to the original buffer on i/o completion, so a callback routine is set up to do this, while for write, the data is copied from the original buffer to the bounce buffer prior to issuing the operation. Since an original buffer may be in a high memory area that's not mapped in kernel virtual addr, a kmap operation may be required for performing the copy, and special care may be needed in the completion path as it may not be in irq context. Special care is also required (by way of GFP flags) when allocating bounce buffers, to avoid certain highmem deadlock possibilities.

It is also possible that a bounce buffer may be allocated from high-memory area that's not mapped in kernel virtual addr, but within the range that the device can use directly; so the bounce page may need to be kmapped during copy operations. [Note: This does not hold in the current implementation, though]

There are some situations when pages from high memory may need to be kmapped, even if bounce buffers are not necessary. For example a device may need to abort DMA operations and revert to PIO for the transfer, in which case a virtual mapping of the page is required. For SCSI it is also done in some scenarios where the low level driver cannot be trusted to handle a single sg entry correctly. The driver is expected to perform the kmaps as needed on such occasions using the __bio_kmap_atomic and bio_kmap_irq routines as appropriate. A driver could also use the blk_queue_bounce() routine on its own to bounce highmem i/o to low memory for specific requests if so desired.

iii. The i/o scheduler algorithm itself can be replaced/set as appropriate

As in 2.4, it is possible to plugin a brand new i/o scheduler for a particular queue or pick from (copy) existing generic schedulers and replace/override certain portions of it. The 2.5 rewrite provides improved modularization of the i/o scheduler. There are more pluggable callbacks, e.g for init, add request, extract request, which makes it possible to abstract specific i/o scheduling algorithm aspects and details outside of the generic loop. It also makes it possible to completely hide the implementation details of the i/o scheduler from block drivers.

I/O scheduler wrappers are to be used instead of accessing the queue directly. See section 4. The I/O scheduler for details.

1.2 Tuning Based on High level code capabilities

i. Application capabilities for raw i/o

This comes from some of the high-performance database/middleware requirements where an application prefers to make its own i/o scheduling decisions based on an understanding of the access patterns and i/o characteristics

ii. High performance filesystems or other higher level kernel code's capabilities

Kernel components like filesystems could also take their own i/o scheduling decisions for optimizing performance. Journalling filesystems may need some control over i/o ordering.

What kind of support exists at the generic block layer for this ?

The flags and rw fields in the bio structure can be used for some tuning from above e.g indicating that an i/o is just a readahead request, or for marking barrier requests (discussed next), or priority settings (currently unused). As far as user applications are concerned they would need an additional mechanism either via open flags or ioctls, or some other upper level mechanism to communicate such settings to block.

1.2.1 I/O Barriers

There is a way to enforce strict ordering for i/os through barriers. All requests before a barrier point must be serviced before the barrier request and any other requests arriving after the barrier will not be serviced until after the barrier has completed. This is useful for higher level control on write ordering, e.g flushing a log of committed updates to disk before the corresponding updates themselves.

A flag in the bio structure, BIO_BARRIER is used to identify a barrier i/o. The generic i/o scheduler would make sure that it places the barrier request and all other requests coming after it after all the previous requests in the queue. Barriers may be implemented in different ways depending on the driver. For more details regarding I/O barriers, please read barrier.txt in this directory.

1.2.2 Request Priority/Latency

Todo/Under discussion:
Arjan's proposed request priority scheme allows higher levels some broad control (high/med/low) over the priority of an i/o request vs other pending requests in the queue. For example it allows reads for bringing in an executable page on demand to be given a higher priority over pending write requests which haven't aged too much on the queue. Potentially this priority could even be exposed to applications in some manner, providing higher level tunability. Time based aging avoids starvation of lower priority requests. Some bits in the bi_rw flags field in the bio structure are intended to be used for this priority information.

1.3 Direct Access to Low level Device/Driver Capabilities (Bypass mode)
(e.g Diagnostics, Systems Management)

There are situations where high-level code needs to have direct access to the low level device capabilities or requires the ability to issue commands to the device bypassing some of the intermediate i/o layers.
These could, for example, be special control commands issued through ioctl interfaces, or could be raw read/write commands that stress the drive's capabilities for certain kinds of fitness tests. Having direct interfaces at multiple levels without having to pass through upper layers makes it possible to perform bottom up validation of the i/o path, layer by layer, starting from the media.

The normal i/o submission interfaces, e.g submit_bio, could be bypassed for specially crafted requests which such ioctl or diagnostics interfaces would typically use, and the elevator add_request routine can instead be used to directly insert such requests in the queue or preferably the blk_do_rq routine can be used to place the request on the queue and wait for completion. Alternatively, sometimes the caller might just invoke a lower level driver specific interface with the request as a parameter.

If the request is a means for passing on special information associated with the command, then such information is associated with the request->special field (rather than misuse the request->buffer field which is meant for the request data buffer's virtual mapping).

For passing request data, the caller must build up a bio descriptor representing the concerned memory buffer if the underlying driver interprets bio segments or uses the block layer end*request* functions for i/o completion. Alternatively one could directly use the request->buffer field to specify the virtual address of the buffer, if the driver expects buffer addresses passed in this way and ignores bio entries for the request type involved. In the latter case, the driver would modify and manage the request->buffer, request->sector and request->nr_sectors or request->current_nr_sectors fields itself rather than using the block layer end_request or end_that_request_first completion interfaces.
(See 2.3 or Documentation/block/request.txt for a brief explanation of the request structure fields)

[TBD: end_that_request_last should be usable even in this case;
Perhaps an end_that_direct_request_first routine could be implemented to make handling direct requests easier for such drivers; Also for drivers that

expect bios, a helper function could be provided for setting up a bio corresponding to a data buffer]

<JENS: I dont understand the above, why is end_that_request_first() not usable? Or _last for that matter. I must be missing something>
<SUP: What I meant here was that if the request doesn't have a bio, then
 end_that_request_first doesn't modify nr_sectors or current_nr_sectors,
 and hence can't be used for advancing request state settings on the
 completion of partial transfers. The driver has to modify these fields
 directly by hand.
 This is because end_that_request_first only iterates over the bio list,
 and always returns 0 if there are none associated with the request.
 _last works OK in this case, and is not a problem, as I mentioned earlier
>

1.3.1 Pre-built Commands

A request can be created with a pre-built custom command to be sent directly to the device. The cmd block in the request structure has room for filling in the command bytes. (i.e rq->cmd is now 16 bytes in size, and meant for command pre-building, and the type of the request is now indicated through rq->flags instead of via rq->cmd)

The request structure flags can be set up to indicate the type of request in such cases (REQ_PC: direct packet command passed to driver, REQ_BLOCK_PC: packet command issued via blk_do_rq, REQ_SPECIAL: special request).

It can help to pre-build device commands for requests in advance. Drivers can now specify a request prepare function (q->prep_rq_fn) that the block layer would invoke to pre-build device commands for a given request, or perform other preparatory processing for the request. This is routine is called by elv_next_request(), i.e. typically just before servicing a request. (The prepare function would not be called for requests that have REQ_DONTPREP enabled)

Aside:
  Pre-building could possibly even be done early, i.e before placing the
  request on the queue, rather than construct the command on the fly in the
  driver while servicing the request queue when it may affect latencies in
  interrupt context or responsiveness in general. One way to add early
  pre-building would be to do it whenever we fail to merge on a request.
  Now REQ_NOMERGE is set in the request flags to skip this one in the future,
  which means that it will not change before we feed it to the device. So
  the pre-builder hook can be invoked there.

2. Flexible and generic but minimalist i/o structure/descriptor.

2.1 Reason for a new structure and requirements addressed

Prior to 2.5, buffer heads were used as the unit of i/o at the generic block layer, and the low level request structure was associated with a chain of buffer heads for a contiguous i/o request. This led to certain inefficiencies when it came to large i/o requests and readv/writev style operations, as it forced such requests to be broken up into small chunks before being passed on to the generic block layer, only to be merged by the i/o scheduler

when the underlying device was capable of handling the i/o in one shot.
Also, using the buffer head as an i/o structure for i/os that didn't originate
from the buffer cache unnecessarily added to the weight of the descriptors
which were generated for each such chunk.

The following were some of the goals and expectations considered in the
redesign of the block i/o data structure in 2.5.

i. Should be appropriate as a descriptor for both raw and buffered i/o - avoid cache related fields which are irrelevant in the direct/page i/o path, or filesystem block size alignment restrictions which may not be relevant for raw i/o.
ii. Ability to represent high-memory buffers (which do not have a virtual address mapping in kernel address space).
iii. Ability to represent large i/os w/o unnecessarily breaking them up (i.e greater than PAGE_SIZE chunks in one shot)
iv. At the same time, ability to retain independent identity of i/os from different sources or i/o units requiring individual completion (e.g. for latency reasons)
v. Ability to represent an i/o involving multiple physical memory segments (including non-page aligned page fragments, as specified via readv/writev) without unnecessarily breaking it up, if the underlying device is capable of handling it.
vi. Preferably should be based on a memory descriptor structure that can be passed around different types of subsystems or layers, maybe even networking, without duplication or extra copies of data/descriptor fields themselves in the process
vii. Ability to handle the possibility of splits/merges as the structure passes through layered drivers (lvm, md, evms), with minimal overhead.

The solution was to define a new structure (bio) for the block layer,
instead of using the buffer head structure (bh) directly, the idea being
avoidance of some associated baggage and limitations. The bio structure
is uniformly used for all i/o at the block layer ; it forms a part of the
bh structure for buffered i/o, and in the case of raw/direct i/o kiobufs are
mapped to bio structures.

2.2 The bio struct

The bio structure uses a vector representation pointing to an array of tuples
of <page, offset, len> to describe the i/o buffer, and has various other
fields describing i/o parameters and state that needs to be maintained for
performing the i/o.

Notice that this representation means that a bio has no virtual address
mapping at all (unlike buffer heads).

```
struct bio_vec {
       struct page     *bv_page;
       unsigned short  bv_len;
       unsigned short  bv_offset;
};

/*
 * main unit of I/O for the block layer and lower layers (ie drivers)
 */
```

```
struct bio {
       sector_t            bi_sector;
       struct bio          *bi_next;    /* request queue link */
       struct block_device *bi_bdev;    /* target device */
       unsigned long       bi_flags;    /* status, command, etc */
       unsigned long       bi_rw;       /* low bits: r/w, high: priority */

       unsigned int     bi_vcnt;     /* how may bio_vec's */
       unsigned int     bi_idx;         /* current index into bio_vec array */

       unsigned int     bi_size;     /* total size in bytes */
       unsigned short   bi_phys_segments; /* segments after physaddr coalesce*/
       unsigned short   bi_hw_segments; /* segments after DMA remapping */
       unsigned int     bi_max;      /* max bio_vecs we can hold
                                        used as index into pool */
       struct bio_vec   *bi_io_vec;  /* the actual vec list */
       bio_end_io_t     *bi_end_io;  /* bi_end_io (bio) */
       atomic_t         bi_cnt;      /* pin count: free when it hits zero */
       void             *bi_private;
       bio_destructor_t *bi_destructor; /* bi_destructor (bio) */
};
```

With this multipage bio design:

- Large i/os can be sent down in one go using a bio_vec list consisting
  of an array of <page, offset, len> fragments (similar to the way fragments
  are represented in the zero-copy network code)
- Splitting of an i/o request across multiple devices (as in the case of
  lvm or raid) is achieved by cloning the bio (where the clone points to
  the same bi_io_vec array, but with the index and size accordingly modified)
- A linked list of bios is used as before for unrelated merges (*) - this
  avoids reallocs and makes independent completions easier to handle.
- Code that traverses the req list can find all the segments of a bio
  by using rq_for_each_segment. This handles the fact that a request
  has multiple bios, each of which can have multiple segments.
- Drivers which can't process a large bio in one shot can use the bi_idx
  field to keep track of the next bio_vec entry to process.
  (e.g a 1MB bio_vec needs to be handled in max 128kB chunks for IDE)
  [TBD: Should preferably also have a bi_voffset and bi_vlen to avoid modifying
   bi_offset an len fields]

(*) unrelated merges -- a request ends up containing two or more bios that
    didn't originate from the same place.

bi_end_io() i/o callback gets called on i/o completion of the entire bio.

At a lower level, drivers build a scatter gather list from the merged bios.
The scatter gather list is in the form of an array of <page, offset, len>
entries with their corresponding dma address mappings filled in at the
appropriate time. As an optimization, contiguous physical pages can be
covered by a single entry where <page> refers to the first page and <len>
covers the range of pages (upto 16 contiguous pages could be covered this
way). There is a helper routine (blk_rq_map_sg) which drivers can use to build
the sg list.

Note: Right now the only user of bios with more than one page is ll_rw_kio,

which in turn means that only raw I/O uses it (direct i/o may not work right now). The intent however is to enable clustering of pages etc to become possible. The pagebuf abstraction layer from SGI also uses multi-page bios, but that is currently not included in the stock development kernels. The same is true of Andrew Morton's work-in-progress multipage bio writeout and readahead patches.

2.3 Changes in the Request Structure

The request structure is the structure that gets passed down to low level drivers. The block layer make_request function builds up a request structure, places it on the queue and invokes the drivers request_fn. The driver makes use of block layer helper routine elv_next_request to pull the next request off the queue. Control or diagnostic functions might bypass block and directly invoke underlying driver entry points passing in a specially constructed request structure.

Only some relevant fields (mainly those which changed or may be referred to in some of the discussion here) are listed below, not necessarily in the order in which they occur in the structure (see include/linux/blkdev.h) Refer to Documentation/block/request.txt for details about all the request structure fields and a quick reference about the layers which are supposed to use or modify those fields.

```
struct request {
        struct list_head queuelist;  /* Not meant to be directly accessed by
                                        the driver.
                                        Used by q->elv_next_request_fn
                                        rq->queue is gone
                                        */
        .
        .
        unsigned char cmd[16]; /* prebuilt command data block */
        unsigned long flags;   /* also includes earlier rq->cmd settings */
        .
        .
        sector_t sector; /* this field is now of type sector_t instead of int
                            preparation for 64 bit sectors */
        .
        .

        /* Number of scatter-gather DMA addr+len pairs after
         * physical address coalescing is performed.
         */
        unsigned short nr_phys_segments;

        /* Number of scatter-gather addr+len pairs after
         * physical and DMA remapping hardware coalescing is performed.
         * This is the number of scatter-gather entries the driver
         * will actually have to deal with after DMA mapping is done.
         */
        unsigned short nr_hw_segments;

        /* Various sector counts */
        unsigned long nr_sectors;  /* no. of sectors left: driver modifiable */
        unsigned long hard_nr_sectors;  /* block internal copy of above */
```

```
        unsigned int current_nr_sectors; /* no. of sectors left in the
                                            current segment:driver modifiable */
        unsigned long hard_cur_sectors; /* block internal copy of the above */
        .
        .
        int tag;        /* command tag associated with request */
        void *special;  /* same as before */
        char *buffer;   /* valid only for low memory buffers upto
                         current_nr_sectors */
        .
        .
        struct bio *bio, *biotail;  /* bio list instead of bh */
        struct request_list *rl;
}
```

See the rq_flag_bits definitions for an explanation of the various flags available. Some bits are used by the block layer or i/o scheduler.

The behaviour of the various sector counts are almost the same as before, except that since we have multi-segment bios, current_nr_sectors refers to the numbers of sectors in the current segment being processed which could be one of the many segments in the current bio (i.e i/o completion unit). The nr_sectors value refers to the total number of sectors in the whole request that remain to be transferred (no change). The purpose of the hard_xxx values is for block to remember these counts every time it hands over the request to the driver. These values are updated by block on end_that_request_first, i.e. every time the driver completes a part of the transfer and invokes block end*request helpers to mark this. The driver should not modify these values. The block layer sets up the nr_sectors and current_nr_sectors fields (based on the corresponding hard_xxx values and the number of bytes transferred) and updates it on every transfer that invokes end_that_request_first. It does the same for the buffer, bio, bio->bi_idx fields too.

The buffer field is just a virtual address mapping of the current segment of the i/o buffer in cases where the buffer resides in low-memory. For high memory i/o, this field is not valid and must not be used by drivers.

Code that sets up its own request structures and passes them down to a driver needs to be careful about interoperation with the block layer helper functions which the driver uses. (Section 1.3)

3. Using bios

3.1 Setup/Teardown

There are routines for managing the allocation, and reference counting, and freeing of bios (bio_alloc, bio_get, bio_put).

This makes use of Ingo Molnar's mempool implementation, which enables subsystems like bio to maintain their own reserve memory pools for guaranteed deadlock-free allocations during extreme VM load. For example, the VM subsystem makes use of the block layer to writeout dirty pages in order to be able to free up memory space, a case which needs careful handling. The allocation logic draws from the preallocated emergency reserve in situations where it cannot allocate through normal means. If the pool is empty and it

can wait, then it would trigger action that would help free up memory or replenish the pool (without deadlocking) and wait for availability in the pool. If it is in IRQ context, and hence not in a position to do this, allocation could fail if the pool is empty. In general mempool always first tries to perform allocation without having to wait, even if it means digging into the pool as long it is not less that 50% full.

On a free, memory is released to the pool or directly freed depending on the current availability in the pool. The mempool interface lets the subsystem specify the routines to be used for normal alloc and free. In the case of bio, these routines make use of the standard slab allocator.

The caller of bio_alloc is expected to taken certain steps to avoid deadlocks, e.g. avoid trying to allocate more memory from the pool while already holding memory obtained from the pool.
[TBD: This is a potential issue, though a rare possibility
 in the bounce bio allocation that happens in the current code, since
 it ends up allocating a second bio from the same pool while
 holding the original bio]

Memory allocated from the pool should be released back within a limited amount of time (in the case of bio, that would be after the i/o is completed). This ensures that if part of the pool has been used up, some work (in this case i/o) must already be in progress and memory would be available when it is over. If allocating from multiple pools in the same code path, the order or hierarchy of allocation needs to be consistent, just the way one deals with multiple locks.

The bio_alloc routine also needs to allocate the bio_vec_list (bvec_alloc()) for a non-clone bio. There are the 6 pools setup for different size biovecs, so bio_alloc(gfp_mask, nr_iovecs) will allocate a vec_list of the given size from these slabs.

The bi_destructor() routine takes into account the possibility of the bio having originated from a different source (see later discussions on n/w to block transfers and kvec_cb)

The bio_get() routine may be used to hold an extra reference on a bio prior to i/o submission, if the bio fields are likely to be accessed after the i/o is issued (since the bio may otherwise get freed in case i/o completion happens in the meantime).

The bio_clone() routine may be used to duplicate a bio, where the clone shares the bio_vec_list with the original bio (i.e. both point to the same bio_vec_list). This would typically be used for splitting i/o requests in lvm or md.

3.2 Generic bio helper Routines

3.2.1 Traversing segments and completion units in a request

The macro rq_for_each_segment() should be used for traversing the bios in the request list (drivers should avoid directly trying to do it themselves). Using these helpers should also make it easier to cope with block changes in the future.

```
        struct req_iterator iter;
        rq_for_each_segment(bio_vec, rq, iter)
                /* bio_vec is now current segment */
```

I/O completion callbacks are per-bio rather than per-segment, so drivers that traverse bio chains on completion need to keep that in mind. Drivers which don't make a distinction between segments and completion units would need to be reorganized to support multi-segment bios.

3.2.2 Setting up DMA scatterlists

The blk_rq_map_sg() helper routine would be used for setting up scatter gather lists from a request, so a driver need not do it on its own.

```
        nr_segments = blk_rq_map_sg(q, rq, scatterlist);
```

The helper routine provides a level of abstraction which makes it easier to modify the internals of request to scatterlist conversion down the line without breaking drivers. The blk_rq_map_sg routine takes care of several things like collapsing physically contiguous segments (if QUEUE_FLAG_CLUSTER is set) and correct segment accounting to avoid exceeding the limits which the i/o hardware can handle, based on various queue properties.

- Prevents a clustered segment from crossing a 4GB mem boundary
- Avoids building segments that would exceed the number of physical memory segments that the driver can handle (phys_segments) and the number that the underlying hardware can handle at once, accounting for DMA remapping (hw_segments) (i.e. IOMMU aware limits).

Routines which the low level driver can use to set up the segment limits:

blk_queue_max_hw_segments() : Sets an upper limit of the maximum number of hw data segments in a request (i.e. the maximum number of address/length pairs the host adapter can actually hand to the device at once)

blk_queue_max_phys_segments() : Sets an upper limit on the maximum number of physical data segments in a request (i.e. the largest sized scatter list a driver could handle)

3.2.3 I/O completion

The existing generic block layer helper routines end_request, end_that_request_first and end_that_request_last can be used for i/o completion (and setting things up so the rest of the i/o or the next request can be kicked of) as before. With the introduction of multi-page bio support, end_that_request_first requires an additional argument indicating the number of sectors completed.

3.2.4 Implications for drivers that do not interpret bios (don't handle multiple segments)

Drivers that do not interpret bios e.g those which do not handle multiple segments and do not support i/o into high memory addresses (require bounce buffers) and expect only virtually mapped buffers, can access the rq->buffer field. As before the driver should use current_nr_sectors to determine the size of remaining data in the current segment (that is the maximum it can

transfer in one go unless it interprets segments), and rely on the block layer end_request, or end_that_request_first/last to take care of all accounting and transparent mapping of the next bio segment when a segment boundary is crossed on completion of a transfer. (The end*request* functions should be used if only if the request has come down from block/bio path, not for direct access requests which only specify rq->buffer without a valid rq->bio)

3.2.5 Generic request command tagging

3.2.5.1 Tag helpers

Block now offers some simple generic functionality to help support command queueing (typically known as tagged command queueing), ie manage more than one outstanding command on a queue at any given time.

blk_queue_init_tags(struct request_queue *q, int depth)

Initialize internal command tagging structures for a maximum depth of 'depth'.

blk_queue_free_tags((struct request_queue *q)

Teardown tag info associated with the queue. This will be done automatically by block if blk_queue_cleanup() is called on a queue that is using tagging.

The above are initialization and exit management, the main helpers during normal operations are:

blk_queue_start_tag(struct request_queue *q, struct request *rq)

Start tagged operation for this request. A free tag number between 0 and 'depth' is assigned to the request (rq->tag holds this number), and 'rq' is added to the internal tag management. If the maximum depth for this queue is already achieved (or if the tag wasn't started for some other reason), 1 is returned. Otherwise 0 is returned.

blk_queue_end_tag(struct request_queue *q, struct request *rq)

End tagged operation on this request. 'rq' is removed from the internal book keeping structures.

To minimize struct request and queue overhead, the tag helpers utilize some of the same request members that are used for normal request queue management. This means that a request cannot both be an active tag and be on the queue list at the same time. blk_queue_start_tag() will remove the request, but the driver must remember to call blk_queue_end_tag() before signalling completion of the request to the block layer. This means ending tag operations before calling end_that_request_last()! For an example of a user of these helpers, see the IDE tagged command queueing support.

Certain hardware conditions may dictate a need to invalidate the block tag queue. For instance, on IDE any tagged request error needs to clear both the hardware and software block queue and enable the driver to sanely restart all the outstanding requests. There's a third helper to do that:

```
blk_queue_invalidate_tags(struct request_queue *q)
```

Clear the internal block tag queue and re-add all the pending requests to the request queue. The driver will receive them again on the next request_fn run, just like it did the first time it encountered them.

3.2.5.2 Tag info

Some block functions exist to query current tag status or to go from a tag number to the associated request. These are, in no particular order:

```
blk_queue_tagged(q)
```

Returns 1 if the queue 'q' is using tagging, 0 if not.

```
blk_queue_tag_request(q, tag)
```

Returns a pointer to the request associated with tag 'tag'.

```
blk_queue_tag_depth(q)
```

Return current queue depth.

```
blk_queue_tag_queue(q)
```

Returns 1 if the queue can accept a new queued command, 0 if we are at the maximum depth already.

```
blk_queue_rq_tagged(rq)
```

Returns 1 if the request 'rq' is tagged.

3.2.5.2 Internal structure

Internally, block manages tags in the blk_queue_tag structure:

```
struct blk_queue_tag {
        struct request **tag_index;     /* array or pointers to rq */
        unsigned long *tag_map;         /* bitmap of free tags */
        struct list_head busy_list;     /* fifo list of busy tags */
        int busy;                       /* queue depth */
        int max_depth;                  /* max queue depth */
};
```

Most of the above is simple and straight forward, however busy_list may need a bit of explaining. Normally we don't care too much about request ordering, but in the event of any barrier requests in the tag queue we need to ensure that requests are restarted in the order they were queue. This may happen if the driver needs to use blk_queue_invalidate_tags().

Tagging also defines a new request flag, REQ_QUEUED. This is set whenever a request is currently tagged. You should not use this flag directly, blk_rq_tagged(rq) is the portable way to do so.

3.3 I/O Submission

The routine submit_bio() is used to submit a single io. Higher level i/o routines make use of this:

(a) Buffered i/o:
The routine submit_bh() invokes submit_bio() on a bio corresponding to the bh, allocating the bio if required. ll_rw_block() uses submit_bh() as before.

(b) Kiobuf i/o (for raw/direct i/o):
The ll_rw_kio() routine breaks up the kiobuf into page sized chunks and maps the array to one or more multi-page bios, issuing submit_bio() to perform the i/o on each of these.

The embedded bh array in the kiobuf structure has been removed and no preallocation of bios is done for kiobufs. [The intent is to remove the blocks array as well, but it's currently in there to kludge around direct i/o.] Thus kiobuf allocation has switched back to using kmalloc rather than vmalloc.

Todo/Observation:

A single kiobuf structure is assumed to correspond to a contiguous range of data, so brw_kiovec() invokes ll_rw_kio for each kiobuf in a kiovec. So right now it wouldn't work for direct i/o on non-contiguous blocks. This is to be resolved. The eventual direction is to replace kiobuf by kvec's.

Badari Pulavarty has a patch to implement direct i/o correctly using bio and kvec.

(c) Page i/o:
Todo/Under discussion:

Andrew Morton's multi-page bio patches attempt to issue multi-page writeouts (and reads) from the page cache, by directly building up large bios for submission completely bypassing the usage of buffer heads. This work is still in progress.

Christoph Hellwig had some code that uses bios for page-io (rather than bh). This isn't included in bio as yet. Christoph was also working on a design for representing virtual/real extents as an entity and modifying some of the address space ops interfaces to utilize this abstraction rather than buffer_heads. (This is somewhat along the lines of the SGI XFS pagebuf abstraction, but intended to be as lightweight as possible).

(d) Direct access i/o:
Direct access requests that do not contain bios would be submitted differently as discussed earlier in section 1.3.

Aside:

Kvec i/o:

Ben LaHaise's aio code uses a slightly different structure instead of kiobufs, called a kvec_cb. This contains an array of <page, offset, len> tuples (very much like the networking code), together with a callback function

and data pointer. This is embedded into a brw_cb structure when passed to brw_kvec_async().

Now it should be possible to directly map these kvecs to a bio. Just as while cloning, in this case rather than PRE_BUILT bio_vecs, we set the bi_io_vec array pointer to point to the veclet array in kvecs.

TBD: In order for this to work, some changes are needed in the way multi-page bios are handled today. The values of the tuples in such a vector passed in from higher level code should not be modified by the block layer in the course of its request processing, since that would make it hard for the higher layer to continue to use the vector descriptor (kvec) after i/o completes. Instead, all such transient state should either be maintained in the request structure, and passed on in some way to the endio completion routine.

4. The I/O scheduler

I/O scheduler, a.k.a. elevator, is implemented in two layers. Generic dispatch queue and specific I/O schedulers. Unless stated otherwise, elevator is used to refer to both parts and I/O scheduler to specific I/O schedulers.

Block layer implements generic dispatch queue in block/*.c.
The generic dispatch queue is responsible for properly ordering barrier requests, requeueing, handling non-fs requests and all other subtleties.

Specific I/O schedulers are responsible for ordering normal filesystem requests. They can also choose to delay certain requests to improve throughput or whatever purpose. As the plural form indicates, there are multiple I/O schedulers. They can be built as modules but at least one should be built inside the kernel. Each queue can choose different one and can also change to another one dynamically.

A block layer call to the i/o scheduler follows the convention elv_xxx(). This calls elevator_xxx_fn in the elevator switch (block/elevator.c). Oh, xxx and xxx might not match exactly, but use your imagination. If an elevator doesn't implement a function, the switch does nothing or some minimal house keeping work.

4.1. I/O scheduler API

The functions an elevator may implement are: (* are mandatory)

elevator_merge_fn	called to query requests for merge with a bio
elevator_merge_req_fn	called when two requests get merged. the one which gets merged into the other one will be never seen by I/O scheduler again. IOW, after being merged, the request is gone.
elevator_merged_fn	called when a request in the scheduler has been involved in a merge. It is used in the deadline scheduler for example, to reposition the request if its sorting order has changed.
elevator_allow_merge_fn	called whenever the block layer determines that a bio can be merged into an existing request safely. The io scheduler may still

	want to stop a merge at this point if it results in some sort of conflict internally, this hook allows it to do that.
elevator_dispatch_fn*	fills the dispatch queue with ready requests. I/O schedulers are free to postpone requests by not filling the dispatch queue unless @force is non-zero. Once dispatched, I/O schedulers are not allowed to manipulate the requests - they belong to generic dispatch queue.
elevator_add_req_fn*	called to add a new request into the scheduler
elevator_queue_empty_fn	returns true if the merge queue is empty. Drivers shouldn't use this, but rather check if elv_next_request is NULL (without losing the request if one exists!)
elevator_former_req_fn elevator_latter_req_fn	These return the request before or after the one specified in disk sort order. Used by the block layer to find merge possibilities.
elevator_completed_req_fn	called when a request is completed.
elevator_may_queue_fn	returns true if the scheduler wants to allow the current context to queue a new request even if it is over the queue limit. This must be used very carefully!!
elevator_set_req_fn elevator_put_req_fn	Must be used to allocate and free any elevator specific storage for a request.
elevator_activate_req_fn	Called when device driver first sees a request. I/O schedulers can use this callback to determine when actual execution of a request starts.
elevator_deactivate_req_fn	Called when device driver decides to delay a request by requeueing it.
elevator_init_fn* elevator_exit_fn	Allocate and free any elevator specific storage for a queue.

4.2 Request flows seen by I/O schedulers
All requests seen by I/O schedulers strictly follow one of the following three flows.

```
 set_req_fn ->

 i.   add_req_fn -> (merged_fn ->)* -> dispatch_fn -> activate_req_fn ->
      (deactivate_req_fn -> activate_req_fn ->)* -> completed_req_fn
 ii.  add_req_fn -> (merged_fn ->)* -> merge_req_fn
 iii. [none]
```

-> put_req_fn

4.3 I/O scheduler implementation
The generic i/o scheduler algorithm attempts to sort/merge/batch requests for optimal disk scan and request servicing performance (based on generic principles and device capabilities), optimized for:
i. improved throughput
ii. improved latency
iii. better utilization of h/w & CPU time

Characteristics:

i. Binary tree
AS and deadline i/o schedulers use red black binary trees for disk position sorting and searching, and a fifo linked list for time-based searching. This gives good scalability and good availability of information. Requests are almost always dispatched in disk sort order, so a cache is kept of the next request in sort order to prevent binary tree lookups.

This arrangement is not a generic block layer characteristic however, so elevators may implement queues as they please.

ii. Merge hash
AS and deadline use a hash table indexed by the last sector of a request. This enables merging code to quickly look up "back merge" candidates, even when multiple I/O streams are being performed at once on one disk.

"Front merges", a new request being merged at the front of an existing request, are far less common than "back merges" due to the nature of most I/O patterns. Front merges are handled by the binary trees in AS and deadline schedulers.

iii. Plugging the queue to batch requests in anticipation of opportunities for merge/sort optimizations

Plugging is an approach that the current i/o scheduling algorithm resorts to so that it collects up enough requests in the queue to be able to take advantage of the sorting/merging logic in the elevator. If the queue is empty when a request comes in, then it plugs the request queue (sort of like plugging the bath tub of a vessel to get fluid to build up) till it fills up with a few more requests, before starting to service the requests. This provides an opportunity to merge/sort the requests before passing them down to the device. There are various conditions when the queue is unplugged (to open up the flow again), either through a scheduled task or could be on demand. For example wait_on_buffer sets the unplugging going through sync_buffer() running blk_run_address_space(mapping). Or the caller can do it explicity through blk_unplug(bdev). So in the read case, the queue gets explicitly unplugged as part of waiting for completion on that buffer. For page driven IO, the address space ->sync_page() takes care of doing the blk_run_address_space().

Aside:
This is kind of controversial territory, as it's not clear if plugging is always the right thing to do. Devices typically have their own queues, and allowing a big queue to build up in software, while letting the device be idle for a while may not always make sense. The trick is to handle the fine balance between when to plug and when to open up. Also now that we have

multi-page bios being queued in one shot, we may not need to wait to merge a big request from the broken up pieces coming by.

4.4 I/O contexts

I/O contexts provide a dynamically allocated per process data area. They may be used in I/O schedulers, and in the block layer (could be used for IO statis, priorities for example). See *io_context in block/ll_rw_blk.c, and as-iosched.c for an example of usage in an i/o scheduler.

5. Scalability related changes

5.1 Granular Locking: io_request_lock replaced by a per-queue lock

The global io_request_lock has been removed as of 2.5, to avoid the scalability bottleneck it was causing, and has been replaced by more granular locking. The request queue structure has a pointer to the lock to be used for that queue. As a result, locking can now be per-queue, with a provision for sharing a lock across queues if necessary (e.g the scsi layer sets the queue lock pointers to the corresponding adapter lock, which results in a per host locking granularity). The locking semantics are the same, i.e. locking is still imposed by the block layer, grabbing the lock before request_fn execution which it means that lots of older drivers should still be SMP safe. Drivers are free to drop the queue lock themselves, if required. Drivers that explicitly used the io_request_lock for serialization need to be modified accordingly. Usually it's as easy as adding a global lock:

```
        static DEFINE_SPINLOCK(my_driver_lock);
```

and passing the address to that lock to blk_init_queue().

5.2 64 bit sector numbers (sector_t prepares for 64 bit support)

The sector number used in the bio structure has been changed to sector_t, which could be defined as 64 bit in preparation for 64 bit sector support.

6. Other Changes/Implications

6.1 Partition re-mapping handled by the generic block layer

In 2.5 some of the gendisk/partition related code has been reorganized. Now the generic block layer performs partition-remapping early and thus provides drivers with a sector number relative to whole device, rather than having to take partition number into account in order to arrive at the true sector number. The routine blk_partition_remap() is invoked by generic_make_request even before invoking the queue specific make_request_fn, so the i/o scheduler also gets to operate on whole disk sector numbers. This should typically not require changes to block drivers, it just never gets to invoke its own partition sector offset calculations since all bios sent are offset from the beginning of the device.

7. A Few Tips on Migration of older drivers

Old-style drivers that just use CURRENT and ignores clustered requests, may not need much change. The generic layer will automatically handle clustered requests, multi-page bios, etc for the driver.

For a low performance driver or hardware that is PIO driven or just doesn't support scatter-gather changes should be minimal too.

The following are some points to keep in mind when converting old drivers to bio.

Drivers should use elv_next_request to pick up requests and are no longer supposed to handle looping directly over the request list.
(struct request->queue has been removed)

Now end_that_request_first takes an additional number_of_sectors argument. It used to handle always just the first buffer_head in a request, now it will loop and handle as many sectors (on a bio-segment granularity) as specified.

Now bh->b_end_io is replaced by bio->bi_end_io, but most of the time the right thing to use is bio_endio(bio, uptodate) instead.

If the driver is dropping the io_request_lock from its request_fn strategy, then it just needs to replace that with q->queue_lock instead.

As described in Sec 1.1, drivers can set max sector size, max segment size etc per queue now. Drivers that used to define their own merge functions i to handle things like this can now just use the blk_queue_* functions at blk_init_queue time.

Drivers no longer have to map a {partition, sector offset} into the correct absolute location anymore, this is done by the block layer, so where a driver received a request ala this before:

```
        rq->rq_dev = mk_kdev(3, 5);     /* /dev/hda5 */
        rq->sector = 0;                 /* first sector on hda5 */
```

it will now see

```
        rq->rq_dev = mk_kdev(3, 0);     /* /dev/hda */
        rq->sector = 123128;            /* offset from start of disk */
```

As mentioned, there is no virtual mapping of a bio. For DMA, this is not a problem as the driver probably never will need a virtual mapping. Instead it needs a bus mapping (dma_map_page for a single segment or use dma_map_sg for scatter gather) to be able to ship it to the driver. For PIO drivers (or drivers that need to revert to PIO transfer once in a while (IDE for example)), where the CPU is doing the actual data transfer a virtual mapping is needed. If the driver supports highmem I/O, (Sec 1.1, (ii)) it needs to use __bio_kmap_atomic and bio_kmap_irq to temporarily map a bio into the virtual address space.

8. Prior/Related/Impacted patches

8.1. Earlier kiobuf patches (sct/axboe/chait/hch/mkp)

- orig kiobuf & raw i/o patches (now in 2.4 tree)
- direct kiobuf based i/o to devices (no intermediate bh's)
- page i/o using kiobuf
- kiobuf splitting for lvm (mkp)
- elevator support for kiobuf request merging (axboe)

8.2. Zero-copy networking (Dave Miller)
8.3. SGI XFS - pagebuf patches - use of kiobufs
8.4. Multi-page pioent patch for bio (Christoph Hellwig)
8.5. Direct i/o implementation (Andrea Arcangeli) since 2.4.10-pre11
8.6. Async i/o implementation patch (Ben LaHaise)
8.7. EVMS layering design (IBM EVMS team)
8.8. Larger page cache size patch (Ben LaHaise) and
Large page size (Daniel Phillips)
=> larger contiguous physical memory buffers
8.9. VM reservations patch (Ben LaHaise)
8.10. Write clustering patches ? (Marcelo/Quintela/Riel ?)
8.11. Block device in page cache patch (Andrea Archangeli) - now in 2.4.10+
8.12. Multiple block-size transfers for faster raw i/o (Shailabh Nagar, Badari)
8.13 Priority based i/o scheduler - prepatches (Arjan van de Ven)
8.14 IDE Taskfile i/o patch (Andre Hedrick)
8.15 Multi-page writeout and readahead patches (Andrew Morton)
8.16 Direct i/o patches for 2.5 using kvec and bio (Badari Pulavarthy)

9. Other References:

9.1 The Splice I/O Model - Larry McVoy (and subsequent discussions on lkml, and Linus' comments - Jan 2001)
9.2 Discussions about kiobuf and bh design on lkml between sct, linus, alan et al - Feb-March 2001 (many of the initial thoughts that led to bio were brought up in this discussion thread)
9.3 Discussions on mempool on lkml - Dec 2001.