Improving Linux Block I/O for Enterprise Workloads

Peter Wai Yee Wong, Badari Pulavarty, Shailabh Nagar, Janet Morgan, Jonathan Lahr, Bill Hartner, Hubertus Franke, Suparna Bhattacharya
IBM Linux Technology Center

{wpeter,pbadari,nagar,janetinc,lahr,bhartner,frankeh}@us.ibm.com, bsuparna@in.ibm.com
http://lse.sourceforge.net/

Abstract

The block I/O subsystem of the Linux kernel is one of the critical components affecting the performance of server workloads. Servers typically scale their I/O bandwidth by increasing the number of attached disks and controllers. Hence, the scalability of the block I/O layer is also an important concern.

In this paper, we examine the performance of the 2.4 Linux kernel's block I/O subsystem on enterprise workloads. We identify some of the major bottlenecks in the block layer and propose kernel modifications to alleviate these problems in the context of the 2.4 kernel. The performance impact of the proposed patches is shown using a decision-support workload, a microbenchmark, and profiling tools. We also examine the newly rewritten block layer of the 2.5 kernel to see if it addresses the performance bottlenecks discovered earlier.

1 Introduction

Over the past few years, Linux has made remarkable progress in becoming a server operating system. The release of Version 2.4 of the Linux kernel has been heralded as helping Linux break the enterprise barrier [5]. Since then, the kernel developer community has redoubled its efforts in improving the scalability of Linux on a variety of server platforms. All major server vendors such as IBM, HP, SGI, Compaq, Dell and Sun not only support Linux on their platforms, but are investing a considerable effort in improving Linux's enterprise capabilities. The Linux Technology Center (LTC) of IBM, in particular, has been a major contributor in improving Linux kernel performance and scalability. This paper highlights the efforts of the LTC in improving the performance and scalability of the block I/O subsystem of the Linux kernel.

Traditionally, the kernel block I/O subsystem has been one of the critical components affecting server workload performance. While I/O hardware development has made impressive gains in increasing disk capacity and reducing disk size, there is an increasing gap between disk latencies and processor speeds or memory access times. Disk accesses are slower than memory accesses by two orders of magnitude. Consequently, servers running I/O intensive workloads need to use large numbers of disks and controllers to provide sufficient I/O bandwidth to enterprise applications. In such environments, the kernel's block I/O layer faces a twofold challenge: it must scale well with a large number of I/O devices and it must minimize the kernel overhead for each I/O transfer.

This paper examines how the Linux kernel's block I/O subsystem handles these twin goals of scalability and performance. Using version 2.4.17 of the kernel as a baseline, we systematically identify I/O performance bottlenecks using kernel profiling tools. We propose solutions in the form of kernel patches, all but one of which has been developed by the authors. The performance improvements resulting from these patches are presented using a decision-support workload, a disk I/O microbenchmark and profiling data. In brief, the I/O performance bottlenecks addressed are as follows:

- **Avoiding the use of bounce buffers**: The kernel can directly map only the first gigabyte of physical memory. I/O to high memory (beyond 1 GB) is done through buffers defined in low memory and involves an extra copy of the data being transferred. Capitalizing on the ability of PCI devices to directly address all 4GB, the block-highmem patch written by Jens Axboe can circumvent the need to use bounce buffers.

- **Splitting the I/O request lock**: Each I/O device in the system has an associated request queue which provides ordering and memory resources for managing I/O requests to the device. In the 2.4 kernel, all I/O request queues are protected by a single `io_request_lock` which can be highly contended on SMP machines with multiple disks and a heavy I/O load. We propose a solution that effectively replaces the io_request_lock with per queue locks.

- **Page-sized raw I/O transfers**: Raw I/O, which refers to unbuffered I/O done through the `/dev/raw` interface, breaks I/O requests into 512-byte units (even if the device hardware and associated driver is capable of handling larger requests). The 512-byte requests end up being recombined within the request queue before being processed by the device driver. We present an alternative that permits raw I/O to be done at a page-size granularity.

- **Efficient support for vector I/O**: I/O intensive applications often need to perform vector (scatter/gather) raw I/O operations which transfer a contiguous region on disk to discontiguous memory regions in the application's address space. The Linux kernel currently handles vectored raw I/O by doing a succession of blocking I/O operations on each individual element of the I/O vector. We implement efficient support for vector I/O by allowing the vector elements to be processed together as far as possible.

- **Lightweight kiobufs**: The main data structure used in raw I/O operations is the kiobuf. As defined in 2.4.17, the kiobuf data structure is very large. When raw I/O is performed on a large number of devices, the memory consumed by kiobufs is prohibitive. We demonstrate a simple way to reduce the size of the kiobuf structure and allow more I/O devices to be used for a given amount of system memory.

Most of the kernel performance bottlenecks listed above stem from the basic design of the 2.4 block I/O subsystem which relies on buffer heads and kiobufs. The need to maintain compatibility with a large number of device drivers has limited the scope for kernel developers to fix the subsystem as a whole. In the 2.5 development kernel, however, the challenging task of overhauling the block I/O layer has been taken up. One of the goals of the rewrite has been addressing the scalability problems of earlier designs [2]. This paper discusses the new design in light of the performance bottlenecks described earlier.

The rest of the paper is organized as follows. Section 2 presents an overview of the 2.4 kernel block I/O subsystem. The benchmark environment and workloads used are described in Section 3. Sections 4 through 8 describe the performance and resource scalability bottlenecks, proposed solutions and results. The newly written 2.5 kernel block layer is addressed in Section 9. Section 10 concludes with directions for future work.

2 Linux 2.4 Block I/O

For the purpose of this paper, our review of the 2.4 kernel block I/O subsystem will be limited in scope. Specifically, it will focus on the "raw" device interface, which was added by Stephen Tweedie during the Linux 2.3 development series.

Unix® has traditionally provided a raw interface to some devices, block devices in particular, which allows data to be transferred between a user buffer and a device without copying the data through the kernel's buffer cache. This mechanism can boost performance if the data is unlikely to be used again in the short term (during a disk backup, for example), or for applications such as large database management systems that perform their own caching.

To use the raw interface, a device binding must be estabished via the raw command; for example, `raw /dev/raw/raw1 /dev/sda1`. Once bound to a block device, a raw device can be opened just like any other device.

A sampling of the kernel code path for a raw open is as follows:

```
sys_open
```

```
. raw_open
. . alloc_kiovec
```

Notice the call to `alloc_kiovec` to allocate a kernel I/O buffer, also known as a kiobuf. The kiobuf is the primary I/O abstraction used by the Linux kernel to support raw I/O. The kiobuf structure describes the array of pages that make up an I/O operation.

The fields of a kiobuf structure include:

```
// number of pages in the kiobuf
int nr_pages;

// number of bytes in the data buffer
int length;

// offset to first valid byte of the buffer
int offset;

// list of device block numbers for the I/O
ulong blocks[KIO_MAX_SECTORS];

// array of pointers to 1024 pre-allocated
// buffer heads
struct buffer_head * bh[KIO_MAX_SECTORS];

// array of up to 129 page structures,
// one for each page of data in the kiobuf
struct page **  maplist[KIO_STATIC_PAGES];
```

The `maplist` array is key to the kiobuf interface, since functions that operate on pages stored in a kiobuf deal directly with page structures. This approach helps hide the complexities of the virtual memory system from device drivers – a primary goal of the kiobuf interface.

Once the raw device is opened, it can be read and written just like the block device to which it is bound. However, raw I/O to a block device must always be sector aligned, and its length must be a multiple of the sector size. The sector size for most devices is 512 bytes.

Let us examine the code path for a raw device read:

```
sys_read
. raw_read
. . rw_raw_dev
. . . map_user_kiobuf(READ, &mykiobuf,
                        vaddr, len)
```

The result of the call to `map_user_kiobuf()` is that the buffer at virtual address `vaddr` of length `len` is mapped into the kiobuf, and each entry of the kiobuf `maplist[ ]` is set to the page structure for the associated page of data. Note that some or all of the user buffer may first need to be paged into memory:

```
. . . map_user_kiobuf
. . . . find_vma
. . . . handle_mm_fault
```

Once all of the pages of the data buffer are locked in memory, read processing continues with a call to `brw_kiovec()`, where for each sector-size chunk of the data buffer, a pre-allocated buffer head associated with the kiobuf is initialized and passed down to `__make_request`. `__make_request()` calls `create_bounce()` to create a bounce buffer as needed, acquires the `io_request_lock`, and uses buffer head information to merge/enqueue the request onto the device-specific request queue.

```
. . . . brw_kiovec(READ, num_kiobufs=1,
                   &mykiobuf,dev,
                   mykiobuf->blocks,
                   sector_size=512)
. . . . . submit_bh
. . . . . . generic_make_request
. . . . . . make_request(&request_queue,
                         &buff_head)
. . . . . . . . create_bounce
. . . . . . . . generic_plug_device
. . . . . . . . <elevator processing>
. . . . . . . . add_request (enqueue)
. . . . kiobuf_wait_for_io
```

Requests are dequeued when the scheduled `tq_disk` task calls `run_task_queue()` which invokes `generic_unplug_device()`. In the case of SCSI, `generic_unplug_device()` invokes `scsi_request_fn()` which dequeues requests and sends them to the driver associated with the request_queue/device.

```
. . . . run_task_queue
. . . . . generic_unplug_device
. . . . . . q->request_fn (scsi_request_fn)
. . . . . . . blkdev_dequeue_request (dequeue)
. . . . . . . scsi_dispatch_cmd
```

The `read()` system call returns once the I/O has completed; that is, after all buffer heads associated with the kiobuf have been processed for completion.

3 Workload and experimental setup

We have been using a decision support benchmark and a disk I/O microbenchmark to study the performance of block I/O. The decision support workload (henceforth called DSW) consists of a suite of highly complex queries accessing a 30GB database. We use IBM DB2® UDB 7.2 as the database management system.

The disk I/O microbenchmark (henceforth called DM) is a multi-threaded disk test. There are a total of 32 raw devices which are mapped to 32 physical disks. DM creates 32 processes. For the read test, each process issues 4096 reads of 64KB each to a raw device. The readv test issues the same number of reads, but uses 16 iovecs of 4KB each.

For both benchmarks, the system was rebooted before each set of runs. For DSW, each set consisted of a sequence of queries run back to back three times. For DM, each set consisted of the read/readv runs performed back to back three times. We took the average of three runs for the score and CPU utilization.

The benchmarks were run on an 8-way 700MHz Pentium® III machine with 4 GB of main memory. The system used for DSW had a 2 MB L2 cache and 6 RAID controllers. The system used for DM had a 1 MB L2 cache and 4 RAID controllers. Each controller was connected to two storage enclosures with each enclosure containing 10 9.1 GB, 10000 RPM drives. The large number of attached disks allowed a high degree of parallel data access and is typical of the environments in which decision-support workloads are run.

Our baseline (henceforth called Baseline) was Linux 2.4.17 with Ingo Molnar's SMP timer patch applied, plus a number of resource-related changes. In addition, readv was used by the database management system for I/O prefetching. The four main patches discussed in subsequent sections are block-highmem, io_request_lock, rawvary and readv/writev. To measure their performance impact incrementally, we used 4 kernels: SB for Baseline+block-highmem, SBI for SB+io_request_lock, SBIR for SBI+rawvary and SBIRV for SBIR+readv/writev.

As a first step towards identifying I/O bottlenecks, the Baseline kernel was profiled using the Kernprof tool [4]. Table 1 shows the percentage of time spent in the most time-consuming kernel functions running a DSW query on the Baseline kernel. We see that `bounce_end_io_read()` is the most expensive function of non-idle time. This function is used when the kernel performs I/O using bounce buffers. The problem caused by bounce buffers and its resolution is described in the next section.

Kernel Function	% Total Time
default_idle	52
bounce_end_io_read	8
do_softirq	7
tasklet_hi_action	6
__make_request	3

Table 1: Profiling data showing percentage of time spent in different kernel functions while running a DSW query on the Baseline kernel.

4 Avoiding the use of bounce buffers

To explain the bounce buffer problem we first take a look at how the Linux 2.4 kernel addresses physical memory. The discussion assumes an x86 architecture though most of the concepts apply to all 32-bit systems. The 4 GB address space defined by 32 bits is divided into two parts: a user virtual address space (0-3GB) and a kernel virtual address space (3-4GB). The physical memory of a system (which is not limited to 4 GB) is divided into three zones:

- DMA Zone (0-16 MB): ISA cards with only 24-bit DMA space use this zone.
- Normal Zone (16 MB-896 MB): Memory in this range is directly mapped into the kernel's 1 GB of virtual address space starting at PAGE_OFFSET (normally 0xC0000000).
- High Memory Zone (896 MB-64 GB): Page frames in this zone need an explicit mapping into kernel virtual address space (via the `kmap()` system call) before they can be used by the kernel.

Kernel	Increase in MOI (%)	CPU Utilization (%)		
		user	kernel	idle
Baseline	—	16	43	41
SB	37	22	71	7
SBI	78	41	37	22
SBIR	16	47	34	19
SBIRV	18	55	9	36

Table 2: Performance impact of various patches on the metric of interest (MOI) and CPU utilization for the decisions support workload (DSW). Increases are reported w.r.t the kernel on the previous line.

Kernel	I/O transfer rate		CPU Idle Time	
	Value (MB/s)	Increase (%)	Value (%)	Increase (%)
Using read				
Baseline	54	—	64	—
SB	133	147	21	-68
SBI	235	77	61	192
SBIR	240	2	94	55
2.5.17 kernel	243	—	97	—
Using readv				
SBIR	104	—	41	—
SBIRV	241	132	94	130
2.5.17 kernel	150	—	61	—

Table 3: Performance impact of various patches on the I/O transfer rate and CPU utilization for the disk I/O microbenchmark (DM). Increases are reported w.r.t the kernel on the previous line. Results are also shown for the 2.5.17 kernel.

DMA operations on memory by I/O devices use physical addresses. Since the kernel cannot address high-memory DMA buffers directly while setting up a buffer for DMA, it allocates an area in low memory called the bounce buffer. It then supplies the buffers physical address to the I/O device. Consequently, data transfer between the device and the high-memory target buffer necessitates an extra copy through the bounce buffer. This degrades system performance by using up low memory (for the bounce buffer) and adding the overhead of a memory copy for each I/O transfer.

The bounce buffer is unnecessary for 32-bit PCI devices, which can normally address 4 GB of physical memory directly. Such devices can access high memory directly even though the kernel cannot. The block-highmem patch from Jens Axboe utilizes this property to permit high-memory DMA to occur without the use of bounce buffers.

To make use of the block-highmem patch, most device drivers require a few changes which are documented in the I/O Performance HOWTO [9].

Kernel Function	% Total Time
__make_request	35
default_idle	17
scsi_dispatch_cmd	4
do_ipsintr	4
scsi_request_fn	4

Table 4: Profiling data showing percentage of time spent in different kernel functions while running a DSW query on the SB kernel

The elimination of bounce buffers is illustrated by Table 4 which again shows the most time-consuming kernel functions while running DSW using the SB kernel. Comparing the entries to those shown in Table 1, we find that bounce buffers are no longer being used.

The second row of Table 2 indicates the performance improvement seen by DSW using the block-highmem patch. The metric of interest (MOI) increases by 37%. Similar trends are seen in the performance of DM in Table 3 with the I/O throughput of the read test increasing from 54 MB/s to 133 MB/s (corresponding to a 147% improvement).

Eliminating bounce buffer usage causes another I/O bottleneck to appear. Comparing Tables 4 and 1 we find that `__make_request` is now the most expensive kernel function and the idle time has been reduced from 64% to around 21% under DM, 41% to 7% under DSW. Both these changes are due to the I/O request lock which is the next bottleneck discussed.

5 Splitting the I/O request lock

As mentioned in the last part of the previous section, Tables 1 and 4 indicate a large fraction of time spent in __make_request and a large drop in idle time when DSW is run on `SB`. Using the Lockmeter [3] profiling tool allows us to investigate whether there are any highly contended locks (spinlocks or reader/writer locks). Table 5 shows the lockmeter statistics for the io_request_lock when DSW is run on `SB`. It shows that 66.2% of 8 CPUs are consumed by spinning on the global `io_request_lock` and the function in which the lock sees high contention also corresponds to the most expensive kernel function in Table 1.

The `io_request_lock`, which is a global serialization device, imposes system-wide serialization on enqueuing block I/O requests. The request enqueuing functions use the lock to protect all request queues collectively which means that only one request can be queued at a time.

During normal I/O operations, request queues are accessed and modified by enqueuing and dequeuing functions. Since multiple threads execute these functions, queue integrity must be protected. Code analysis shows that queuing operations on a given queue involve access to queue-specific data, request list anchor (queue_head), request free list (rq), plug state (plugged), but do not require access to data used by queuing operations on other queues. This means that maintaining queue data integrity does not require serialization of queuing to different queues. Queuing operations on different queues are logically independent and can execute concurrently. Of course, multiple queuing operations to the same queue must still be serialized.

To implement concurrent enqueuing, we replaced `io_request_lock` in enqueuing functions with per queue locks (`request_queue.queue_lock`). This serializes enqueuing to the same queue while allowing concurrent enqueuing to different queues. With this change dequeuing functions can no longer rely on `io_request_lock` to serialize with enqueuing functions. To restore this serialization, dequeuing functions were modified to acquire `queue_lock` in addition to `io_request_lock` when accessing queues.

To minimize interlocking between dequeueing and enqueueing functions, we added another level of locks inside dequeueing functions. This allows us to maintain our focus on enqueuing and avoid the impact of further reducing the scope of the `io_request_lock`.

When the above modifications to the generic block I/O code were published for comment, the Linux development community expressed concern about making such major changes to the mature 2.4 kernel. Since the patch modified the locking structure in code which affected all block I/O devices, many viewed the code impact as undesirably pervasive. Unforeseen impacts to other code such as IDE and some device drivers were also pointed out. Since SCSI configurations represent a significant part of our scalability goal and concurrent queuing can be implemented for SCSI without affecting generic i/o code, we decided to isolate SCSI code for our development purposes. Fortunately, the block I/O subsystem provides for such isolation through dynamically assigned I/O queuing functions stored in the request queue and indirectly invoked as function pointers.

To contain code impact within the SCSI subsystem, generic enqueuing and dequeuing functions were copied, renamed, and modified for concurrent queuing. The following generic block I/O (ll_rw_blk.c) functions provided baselines for SCSI functions:

```
__make_request => scsi_make_request
generic_plug_device => scsi_plug_device
generic_unplug_device => scsi_unplug_device
get_request => scsi_get_request
get_request_wait => scsi_get_request_wait
blk_init_queue => scsi_init_queue
```

Concurrent queuing is activated for all devices under an adapter driver by setting the new `concurrent_queue` field of the `Scsi_Host_Template` structure used for driver registration. This allows

Kernel function holding lock	Lock Utilization (%)	Mean Lock Hold Time (μs)	Lock Spin Time Mean (μs)	Lock Spin Time % CPU	Number of lock acquisitions
`All spinlocks`		3.7	62.0	66.8	68774051
io_request_lock	50.2	5.2	65.0	66.2	15640659
. __make_request	23.5	3.8	64.0	42.8	9973270
. do_ipsintr	8.3	20.0	66.0	3.1	660212
. scsi_dispatch_cmd	6.8	13.0	66.0	3.9	877838
. generic_unplug_device	4.5	8.8	65.0	3.2	835530

Table 5: Lockmeter data for io_request_lock with DSW on the SB kernel.

control over which drivers use concurrent queuing and preserves original request queuing behavior by default. Drivers which enable concurrent queuing must protect any request queue access with queue locks.

With the application of the `io_request_lock` patch (IORL), the MOI of DSW improves by 78% over the baseline `SB`, as is seen in row three of Table 2. The transfer rate of DM also increases significantly from 133 MB/sec to 235 MB/sec (Table 3). Note that there is a significant increase of idle time in both cases due to the reduction of the spin time. Table 6 verifies that the lock contention seen by the io_request_lock has been reduced. `scsi_make_request()` is shown using a per-queue lock and the aggregate contention on the per-queue locks is reduced as well.

Kernel Function	% Total Time
default_idle	41
schedule	4
ips_make_passthru	4
tasklet_hi_action	3
do_softirq	3
brw_kiovec	3
scsi_back_merge_fn_dc	3
scsi_release_buffers	3
scsi_back_merge_fn_	2
scsi_dispatch_cmd	2
end_buffer_io_kiobuf	2

Table 7: Kernprof data for DSW on the SBI kernel.

Table 7 lists the most expensive kernel functions for DSW running on `SBI`. A significant fraction of kernel time is spent in `brw_kiovec()` and many SCSI mid-layer functions. One reason for that is the use of 512-byte blocks for raw I/O as explained in the next section.

6 Raw I/O optimization patch

This section provides information on the optimization patch that we developed to increase the block size used for raw I/O. The patch can significantly improve CPU utilization by reducing the number of buffer heads needed for such operations.

As explained in Section 2, `rw_raw_dev` calls `map_user_kiobuf` to map the user buffer into a kiobuf, and then invokes `brw_kiovec` to submit the I/O. `brw_kiovec` breaks up each mapped page into sector-size pieces (normally 512 bytes) and passes them one at a time to `make_request`. Each sector-size piece is represented using one of the 1024 pre-allocated buffer heads associated with the kiobuf. Assuming a sector-size of 512 bytes, `brw_kiovec` would use 512 buffer heads and invoke `make_request` 512 times to process a 256K raw read or write.

`make_request` uses the buffer head information to enqueue the request on the device-specific request queue and returns to `brw_kiovec`. When the lesser of all mapped pages or `KIO_STATIC_PAGES` of the kiobuf have been processed in this way, `brw_kiovec` calls `kiobuf_wait_for_io`. `kiobuf_wait_for_io` returns after the I/O completion routine has been called for all of the mapped buffer heads of the kiobuf.

While the block I/O subsystem will normally merge buffer heads into larger requests, there is still overhead incurred with each buffer head. For example, the interrupt handler for the block device must invoke the `b_end_io` method for each buffer head at I/O completion. The second column of Table 8 shows function call frequencies in a call graph trace for 128 reads of 128KB each using a 512-byte block size. The large number of calls to `submit_bh()` indicates the severity of the problem.

Kernel function holding lock	Lock Contention (%)	Mean Lock Hold Time (μs)	Lock Spin Time		Number of lock acquisitions
			Mean (μs)	% CPU	
`All spinlocks`		2.1	15.0	13.9	63777886
io_request_lock	39.6	8.7	32.0	7.6	2490263
. do_ipsintr	16.3	26.0	32.0	1.4	339486
. scsi_unplug_device	11.7	18.0	32.0	1.2	357540
. scsi_dispatch_cmd	8.4	13.0	31.0	1.4	363421
scsi_make_request	15.3	0.9	13.0	0.3	9520872

Table 6: Lockmeter data showing benefits of the IORL patch for DSW on the SBI kernel.

Kernel function	Frequency	
	Baseline	Baseline+rawvary
sys_read	138	138
. raw_read	128	128
. . rw_raw_dev	128	128
. . . brw_kiovec	128	128
. . . . submit_bh	32768	4096
. . . . . generic_make_request	32789	4160
. . . . . . _make_request	32789	4160
. . . . elevator_linus_merge	32659	4029
. . . . scsi_back_merge_fn_c	32641	4013

Table 8: Reduction in frequencies of function calls using the rawvary patch for 128 reads of 128KB each.

The patch we developed can reduce 8-fold the number of buffer heads required for a raw I/O operation. This was accomplished by changing `brw_kiovec` to break up the user buffer into sector-size pieces only until the buffer address is aligned on a page boundary. Once properly aligned, the remainder of the mapped pages are submitted to `make_request` with a block size (`b_size`) of 4 KB instead of sector-size. Note that the last buffer head may have a `b_size` which is neither sector-size nor 4 KB depending on the total length of the I/O request.

Since we could not practically determine whether a given device driver can support buffer heads of variable-block sizes in a merged request, the patch enables the optimization for the Adaptec, Qlogic SCSI and IBM ServeRAID drivers only. Other drivers can make use of the patch by setting the `can_do_varyio` bit in the `Scsi_Host_Template` structure before calling `scsi_register`.

The third column of Table 8 highlights the reduction in kernel overhead as a result of using the patch. The number of calls to `submit_bh` are reduced by a factor of 8. The MOI of DSW improved by 16% over SBI, as seen in the fourth row of Table 2. The transfer rate of DM also increased slightly from 235 MB/sec to 240 MB/sec (Table 3). However, there was an improvement of 55% in the idle time.

The raw I/O optimization patch, also known as the rawvary patch, has been integrated into Andrea Arcangeli's 2.4.18pre7aa2 kernel and Alan Cox's 2.4.18pre9-ac2 kernel.

7 Efficient support for vector I/O

Scatter-gather I/O is needed by an application when it needs to transfer data between a contiguous portion of a disk file and non-contiguous memory buffers in its address space. Typically this is done by invoking the `readv()/writev()` system calls and passing an array of `struct iovec` entries. Each `iovec` entry represents a contiguous memory buffer of length `iov_len` located at `iov_base`. This entry is henceforth called an *iochunk* since the kernel does not define a distinct name for it and the term iovec suggests an array rather than an individual element. To simplify the discussion, we refer only to the readv operation. For raw I/O operations, writev differs mainly in the direction of data transfer.

In the 2.4 kernel, the readv system call using a file descriptor is implemented by calling the corresponding file's readv function. When there is no readv function exported, as is the case for raw I/O, the kernel defaults to using repeated invocations of the file's read function which is always defined. Each iochunk of the iovec leads to a separate blocking read being performed. This imposes a dual penalty on the application. It imposes the overhead of multiple calls to various functions in the entire I/O processing path from the top level `sys_readv()` down to the SCSI layer elevator and merging functions. Worse, it serializes the I/O requests seen by the low-level device driver. Since a separate read/write is performed for each iochunk and these calls block until I/O completes, the kernel's ability to take advantage of large DMA operations is severely limited. The elevator code invoked by the `make_request()` function cannot merge requests from different iochunks and hence the SCSI device driver cannot create large scatter-gather lists for the controller.

To reduce this inefficiency, we created a patch defining readv and writev functions for raw devices. The functions operate in two phases while processing an iovec. In the first phase, they map the pages of several iochunk buffers into a single kiobuf. The number of pages mapped to a single kiobuf is limited by the KIO_STATIC_PAGES limit (which is 65 when the system page size is 4 KB). Once this limit is reached (or if the entire iovec has been mapped), `brw_kiovec()` is invoked to submit the I/O represented by the kiobuf. As explained in Section 2, `brw_kiovec()` is a blocking function that returns only when the corresponding I/O is complete or if there is an error. The two phases are repeated until all iochunks of the iovec are processed.

The patch relies upon one important modification to `struct kiobuf`. As explained in Section 2, `struct kiobuf` has only one offset and length field. The offset field represents the offset into the (virtual) memory buffer. When the pages of multiple memory buffers are mapped in to the same kiobuf, we need a per-page offset and length information. We modified `struct kiobuf` to add this information using the following structure:

```
struct pinfo
{
        int poffset[KIO_STATIC_PAGES];
        int plen[KIO_STATIC_PAGES];
};
```

```
struct kiobuf
{
:
:
struct pinfo *  pinfo;
}
```

There are other approaches to providing readv/writev support. In an earlier attempt, we tried to map an iovec onto a `kiovec` consisting of multiple kiobufs. However, that approach increased memory consumption since `struct kiobuf` is quite heavyweight and also because the `brw_kiovec()` function only submits I/O for `KIO_STATIC_PAGES` one at a time. Mapping one iochunk onto one kiobuf would have resulted in wasted pointers in the `map_array` without increasing the granularity at which I/O was submitted to the lower layers. Our current approach fits in well with the 2.4 kernel's practice of using only one kiobuf per file. The issue of the heavyweight `struct kiobuf` is discussed in Section 8 though the changes shown there do not warrant reexamining our choice to map multiple iochunks into a single kiobuf.

Using the readv/writev patch improves the MOI of DSW by 18% (Table 2) and the I/O transfer rate of DM from 104 MB/s to 241 MB/s (Table 3). CPU utilization also decreases significantly for both cases.

8 Lightweight kiobufs

In the 2.4.17 kernel, a kiobuf is allocated for each raw device open. The allocated kiobuf is saved in the `f_iobuf` field of the `file` object for the device special file and is used for doing reads/writes on the raw device. Each kiobuf is 8792 bytes in size and is allocated from `vmalloc()` space which is generally 128 MB. Middleware such as database managers often keep a large number of files open. For raw I/O, the number of open calls generally scales with the number of devices (which are accessed through device special files). In such cases, a heavyweight kiobuf is a drain on the kernel's low memory in general and `vmalloc` space in particular.

To enable a large number of raw devices to be opened simultaneously, we modified the kiobuf structure to reduce its memory footprint. Much of the memory consumed by a kiobuf is due to the two arrays:

```
struct buffer_head * bh[KIO_MAX_SECTORS];
unsigned long blocks[KIO_MAX_SECTORS];
```

With `KIO_MAX_SECTORS` being 1024, these arrays consume 8192 bytes.

We changed the kiobuf structure as follows:

1. The buffer head array `bh` was replaced by a linked list. To link the various buffer heads of a kiobuf together, we used the `b_next_free` field of struct buffer_head. This field is not used in buffer-head processing in the raw I/O path.
2. The `blocks` array was replaced by a single number. Normally, the `blocks` array contains the physical disk block numbers corresponding to the logical blocks of a file. For accesses which don't go through a filesystem, the logical and physical disk blocks are the same. Hence, for raw I/O, the blocks array contains sequential numbers. We replaced the blocks array by a single number indicating the starting disk block and modified the code doing raw I/O to generate the remaining sequence of disk block numbers.

Together these modifications reduced the size of the kiobuf to 608 bytes and allowed them to be allocated using `kmalloc()` instead of `vmalloc()`.

A further reduction in the memory footprint of the kiobuf was enabled by the use of the rawvary patch described in Section 6. Since I/O is done 4KB at a time, a kiobuf needs only `KIO_STATIC_PAGES` (65) buffer heads instead of `KIO_MAX_SECTORS` (1024) to represent the maximum I/O that can be done using a single kiobuf.

9 2.5 changes - tackling the root of the problem ?

In part, the block layer rewrite in 2.5 was motivated by some of the well known shortcomings of the 2.4 block layer that we came across in the earlier sections. Of major concern was the suboptimal performance and resource overhead in the case of large I/O requests, I/O on high memory addresses, and I/O operations that do not originate directly from the buffer cache like raw/direct I/O and page I/O.

Most of these problems stemmed from the use of the buffer head as the unit of I/O at the generic block layer, and the basic limitations on the size and nature of I/O buffers that could be represented by a single buffer head. It could only be a contiguous chunk at a virtually mapped address, of size one blocksize unit, which could not exceed a page and had to be aligned at a block boundary (as per the block size used). This led to the described inefficiencies in handling large I/O requests and readv/writev style operations, as it forced such requests to be broken up into small chunks so that they could be mapped to buffer heads before being passed on one by one to the generic block layer, only to be merged back by the I/O scheduler when the underlying device is 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.

At the same time, one of the good things about the original design was that splitting and merging of requests was a simple matter of breaking or chaining pointers, without requiring any memory allocation or move.

In the context of raw or direct I/O, a second aspect of concern was the weighty nature of the higher level kiobuf data structure as discussed in earlier sections. One of the shortcomings of the kiobuf is that a single kiobuf can represent only a contiguous user address range, which makes it unsuitable for user space memory vectors of the form supplied by readv/writev. While arrays of kiobufs, namely kiovecs, are defined, they are too heavyweight for use in readv/writev.

Another crucial issue addressed in the rewrite was the matter of the single global I/O request lock bottleneck, especially in the case of independent/parallel I/Os to multiple disks.

9.1 The origin of BIO

The solution implemented in 2.5 by Jens Axboe [2] addresses these inefficiencies at a fundamental level by defining some new data structures. A flexible structure called BIO has been created for the block layer instead of using the buffer head structure directly, thus eliminating any associated baggage and restrictions. The abstraction is sector oriented and is unaware of filesystem block sizes.

The BIO structure uses a generic 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. The core memory vector representation is capable of describing a set of non-page aligned fragments in a uniform manner across various layers including zero copy network I/O, and kernel asynchronous I/O [1]. This makes it possible for the same descriptor to be passed across subsystems and be useful for things like streaming I/O from network to disk and vice-versa. Such a descriptor can directly refer to user space buffers in a process context independent way, and forms an I/O currency similar to that proposed in [7].

The new scheme enables large, as well as vectored I/Os, to be described as a single unit within the limits of the device capabilities and is adequate for specifying high memory buffers as well since it doesn't require a virtual address mapping. The underlying DMA mapping functions have been modified to work with this representation. Bounce buffers become necessary only where the device does not support I/O into high memory buffers. In situations where the driver needs to access the buffer by virtual address, it performs a temporary kmap (e.g. if falling back to PIO in IDE).

A low level request structure may consist of a chain of BIOs (potentially arising from multiple sources or callers) for a contiguous area on disk, a concept which retains some of the goodness of the original design in terms of ease of request merging, and treatment of individual completion units. The BIO structure maintains an index into the vector to help keep track of which fragments have been transferred so far, in case the transfer or a subsequent copy happens in stages. Notice also, that potentially, a single entry in the vector could describe a fragment greater than a page size, i.e. across contiguous physical (or perhaps more accurately, logical) pages. Splitting an I/O request involves cloning the BIO structure and adjusting the indices to cover the desired portions of the original vector.

Using a separate structure introduces a level of allocation and setup in some cases as a BIO has to be constructed for each I/O (e.g. rather than directly utilizing a bh in the case of buffered I/O). Typically BIOs are allocated from a designated BIO mempool, where mempool refers to Ingo Molnar's new memory pool infrastructure in 2.5. The allocation scheme is designed to avoid deadlocks as in a scenario when the I/O in question is a writeout issued under memory pressure. A caller avoids possibilities of holding on to a BIO without initiating any action (like starting low level I/O) that would eventually recycle it back to the pool. The situation gets tricky if further BIO allocations become necessary in order to proceed with the request (e.g. a bounce BIO in situations where the device doesn't support highmem I/O, or BIO allocations required for splitting the I/O in the case of lvm/md/evms). To avoid any possibility of a deadlock, multiple allocations held at a time from the same pool by a thread ought to be atomic or pipelined. Alternatively, the allocations could be spread across multiple pools in an established order.

9.2 Elimination of IORL

Another major improvement in 2.5 is the removal of the global I/O request lock present in 2.4. Instead, every queue is associated with a pointer to a lock, which is held during queuing. This enables per-queue locks or shared locks across queues depending on the level of concurrency supported by the underlying mid/driver layers. The SCSI midlayer, for example, sets the lock pointer to the same per adapter value for all request queues associated with the devices connected to a given host adapter. Unlike our patched 2.4 SCSI mid-layer which serializes enqueuing per device, this locking scheme serializes at a coarser per adapter granularity.

A notion of command pre-building outside of the queue lock and ahead of request processing by the device has been considered for its potential to improve throughput and interrupt responses, but it has not been explored entirely. Choosing the right moment to prebuild is not trivial - done too early it would require rebuilding on every subsequent merge, done too late, e.g. at the time of actually scheduling a request, it takes up cycles in request processing context which dilutes the desired effect.

9.3 Better per-queue tuning

Improved modularization at the generic block level now enables better per-queue level tuning and consideration of higher level attributes for I/O scheduler performance under specific configurations and workloads. There is support for efficient I/O barriers in cases where corresponding hardware support exists, which could be useful for transaction oriented I/O.

9.4 A job to do - utilizing the framework

At this point, work remains to be done in terms of modifying higher levels in the OS to make optimum use of this new infrastructure. Preliminary experiments running DM show that the 2.5.17 kernel outperforms SBIR for reads but does worse than SBIRV when readv is used (Table 3. This is consistent with the current state of implementation of the new block layer where the readv path has not seen the benefits of the bio structure. In fact, we can even expect a slight degradation for small I/Os because the memory vector structure is inherently a little more complex than the simple virtually mapped buffer in 2.4. For small single segment I/O the drivers end up with an added check for the end of the array, and many of the BIO fields become almost redundant.

Therefore, intelligent pre-merging at higher levels makes sense in this context. A 1:1 mapping between buffer heads and BIOs is not quite efficient. There is ongoing work to rewrite some of the filesystem interfaces to move in this direction. Andrew Morton's multi-page read and writeout patches [8] assemble large BIOs for pagecache pages (for as many corresponding blocks that are contiguous on disk) and submit them directly to the request layer, bypassing buffer heads altogether.

From the perspective of raw/direct I/O, which are the main areas of consideration in the current paper, the relatively heavyweight kiobuf infrastructure would have to be replaced by something like the lighter kvec data structures in Ben LaHaise's asynchronous I/O patches [6], which can support readv/writev operations efficiently.

A `kvec` is pretty close to a bare abstraction of a memory vector array of the form used in a BIO, each tuple of the vector being referred to as a kveclet. It is usually more useful to pass around a `kvec_cb` structure which refers to a `kvec` and its associated callback data for I/O completion purposes.

```
struct kveclet {
   struct page *page;
   unsigned     offset;
   unsigned     length;
}

struct kvec {
   unsigned        max_nr;
   unsigned        nr;
   struct kveclet veclet[0];
}

struct kvec_cb {
   struct kvec *vec;
   void        (*fn)(...);
   void        *data;
}
```

A `kvec` can be mapped to BIO structures for block I/O and similarly to equivalent `skb fragment` structures in the case of network I/O. A single kvec may be split across multiple BIO structures (each pointing to the corresponding section of the `kvec`), each of which acts as a distinct completion unit when more than one low level device requests are involved in serving the I/O. A large user space buffer (especially in the case of vectored I/O), might even be mapped to a big `kvec` a section at a time, and appropriately pipelined for I/O through multiple BIO requests to potentially enhance throughput and latencies for partial completions.

In the case of direct I/O, extents of non-contiguous blocks would have to be mapped to separate BIO units.

There also has been some discussion on the maximum size of BIOs that may be pushed down to the block layer, from the perspective of avoiding chopping up an I/O unless it violates the underlying device limits. Because this decision is more complex than just a matter of absolute size, and may even depend on request queue state, Linus

Torvalds has suggested that drivers could supply a `grow_bio` helper function to handle this. Further complications arise in the case of layered drivers like lvm/md/evms. Andrew Morton has proposed a dynamic `get_max_bytes` interface exported by drivers (cascaded down layered drivers if required), to help build up appropriately sized BIOs to avoid splitting by the lower layers.

Observe that in 2.4 with fixed size (small) buffer heads, the approach was to never split a buffer, but include it as part of the request or create a new request depending on whether it could be fitted within the limits allowable for the device in question. In 2.5, the BIO represents larger variable sizes, having variable number of segments. Such a simplistic approach could result in underutilization of request slots when merging I/Os from different sources. If a buffer exceeds the request size which the device can handle, it breaks up the request. However, splitting up a BIO for a correct fit requires an additional memory allocation. Some points of caution with regard to such allocations at the block layer level have been discussed in an earlier subsection. This is why the question of constructing BIOs of right size arises.

A suitable solution would have to take into account that splitting is expected to be relatively infrequent. Since the general direction is to move towards merging early, `get_max_bytes()` could turn out to be a useful hint even for the corresponding clustering decisions. At the same time, it may not always be feasible or efficient in practice to absolutely guarantee elimination of the need to split I/Os. Thus, a provision for splitting may be required with due caution possibly with a structured use of multiple (layered) mempools and pipelined piecewise submissions to avoid deadlocks.

10 Conclusion and Future Work

In this paper we have highlighted some of the scalability and performance limitations of the 2.4 Linux kernel's block I/O subsystem. Using a decision-support benchmark that is representative of real-world enterprise workloads, we have shown that the 2.4.17 kernel sees I/O related performance bottlenecks when large I/O's are done on raw devices. We systematically investigated these bottlenecks and proposed solutions (as kernel patches) to alleviate them. As a result of using these patches, the decision-support workload sees an 233% improvement in its metric of interest. The benefits of these patches, all but one of which were written by the authors, are further demonstrated through a disk I/O microbenchmark and profiling data.

Most of the problems that we demonstrated are seen because of the use of the buffer head and kiobuf data structures. The new block I/O layer being written for the 2.5 kernel looks very promising as it addresses almost all the problems outlined here. Much work remains to be done to efficiently utilize the new data structures introduced in 2.5. We will continue to actively participate in the kernel community's efforts to improve the performance of both the 2.4 and 2.5 kernels for enterprise workloads.

11 Acknowledgments

We would like to thank the many people on the `lse-tech@lists.sourceforge.net` mailing list who provided us with valuable comments and suggestions during the development of these patches. In particular, we would like to thank Ruth Forester for helping resolve numerous issues with the decision-support workload and Helen Pang for collecting data on the disk I/O microbenchmark. We also appreciate the excellent DB2 performance analysis provided by John Tran, Karen Sullivan and James Cho.

This work was developed as part of the Linux Scalability Effort (LSE) on SourceForge (`sourceforge.net/projects/lse`). All the patches mentioned in this paper can be found in the "I/O Scalability Package" at the LSE site.

This work represents the view of the authors, and does not necessarily represent the view of IBM.

References

[1] Suparna Bhattacharya. Design Notes on Asynchronous I/O (aio) for Linux. http://lse.sourceforge.net/io/aionotes.txt.

[2] Suparna Bhattacharya. Notes on 2.5 Block I/O Layer Changes. http://lse.sourceforge.net/io/bionotes.txt.

[3] R. Bryant and J. Hawkes. Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, Oct 2000.

[4] John Hawkes et. al (Silicon Graphics Inc.). Kernprof. Available at http://oss.sgi.com/projects/kernprof/index.html.

[5] InfoWorld Test Center K. Railsback. Linux 2.4 breaks the enterprise barrier. http://www.infoworld.com/articles/tc/xml/01/01/15/010115tclinux.xml.

[6] Benjamin LaHaise. Kernel Asynchronous I/O Patches. http://www.kvack.org/ blah/aio.

[7] Larry McVoy. The Splice I/O Model.

[8] Andrew Morton. Multi-page writeout and readahead patch. http://www.zip.com.au/ akpm/linux/patches/2.5/2.5.8.

[9] Sharon Snider. I/O Performance HOWTO. http://www.tldp.org/HOWTO/IO-Perf-HOWTO/index.html.

Trademarks