

[rfc] direct IO submission and completion scalability issues

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-07/msg12153.html>

- *From:* "Siddha, Suresh B" <suresh.b.siddha@xxxxxxxx>
 - *Date:* Fri, 27 Jul 2007 18:21:28 -0700
-

We have been looking into the linux kernel direct IO scalability issues with database workloads. Comments and suggestions on our below experiments are welcome.

In the linux kernel, direct IO requests are not batched at the block layer. i.e, as a new request comes in, the request get directly submitted to the IO controller on the same cpu that the request originates. And the IO completion likely happens on a different cpu which is processing interrupts. This results in cacheline bouncing of some of the hot kernel cachelines (like timers, scsi cmds, slab, sched, etc) and is becoming an important scalability issue as the number of cpus and distance between them increase with multi-core and numa.

In case of the controllers which support RIO/ZIO modes (like some qla2xxx), IO submission path on each cpu also checks if there any completed IO commands in the response queue and triggers softirq on the same cpu to process the completed commands. This results in each logical cpu in the system spending sometime in softirq processing and this causes contentions in spinlocks and other data structures.

Not sure when the IO controllers with multiple request/response queues will be available in the market. In that case we can dedicate each queue pair to group of cpus(/a node) and be done with this problem.

In the absence of such HW today, we were looking into possible solutions for these problems and did couple of experiments as part of this.

In the first experiment, we removed the completed IO command processing during IO submission. This will now result in the processing of IO commands only on the cpu receiving interrupts. This will result in more interrupts (as we are not doing any proactive processing) but wanted to see if this is a win over each cpu doing the softirq processing. This gave a 1.36% performance improvement on a x86_64 MP system (total 16 logical cpus) and on two node ia64 platform(2 nodes, 8 cores, 16 threads) we got 1.5% improvement [please look at observation #1 below].

Reference patch for this:

[rfc] direct IO submission and completion scalability issues

```
diff --git a/drivers/scsi/qla2xxx/qla_iocb.c b/drivers/scsi/qla2xxx/qla_iocb.c
index c5b3c61..357a497 100644
--- a/drivers/scsi/qla2xxx/qla_iocb.c
+++ b/drivers/scsi/qla2xxx/qla_iocb.c
@@ -414,11 +414,6 @@ qla2x00_start_scsi(srb_t *sp)
WRT_REG_WORD(ISP_REQ_Q_IN(ha, reg), ha->req_ring_index);
RD_REG_WORD_RELAXED(ISP_REQ_Q_IN(ha, reg)); /* PCI Posting. */

- /* Manage unprocessed RIO/ZIO commands in response queue. */
- if (ha->flags.process_response_queue &&
- ha->response_ring_ptr->signature != RESPONSE_PROCESSED)
- qla2x00_process_response_queue(ha);
-
spin_unlock_irqrestore(&ha->hardware_lock, flags);
return (QLA_SUCCESS);

@@ -844,11 +839,6 @@ qla24xx_start_scsi(srb_t *sp)
WRT_REG_DWORD(&reg->req_q_in, ha->req_ring_index);
RD_REG_DWORD_RELAXED(&reg->req_q_in); /* PCI Posting. */

- /* Manage unprocessed RIO/ZIO commands in response queue. */
- if (ha->flags.process_response_queue &&
- ha->response_ring_ptr->signature != RESPONSE_PROCESSED)
- qla24xx_process_response_queue(ha);
-
spin_unlock_irqrestore(&ha->hardware_lock, flags);
return QLA_SUCCESS;
```

Observation #1: This experiment puts heavy load on the cpu processing interrupts. As such, equal distribution of task load by the scheduler didn't give expected performance improvement(as cpu's with no interrupts race to idle and migrate some tasks during idle balance, leading to some increase in idle time aswell as costs associated with excessive task migration). We tweaked our manual task binding so that cpu's with no interrupts get proportionally more load compared to cpu's which process interrupts and this gave a nice performance boost as mentioned above. Perhaps, we need to make the scheduler load balancing aware of the irq load on that cpu.

Second experiment which we did was migrating the IO submission to the IO completion cpu. Instead of submitting the IO on the same cpu where the request arrived, in this experiment the IO submission gets migrated to the cpu that is processing IO completions(interrupt). This will minimize the access to remote cachelines (that happens in timers, slab, scsi layers). The IO submission request is forwarded to the kblockd thread on the cpu receiving the interrupts. As part of this, we also made kblockd thread on each cpu as the highest priority thread, so that IO gets submitted as soon as possible on the interrupt cpu with out any delay. On x86_64 SMP platform with 16 cores, this resulted in 2% performance improvement and 3.3% improvement on two node ia64 platform.

Quick and dirty prototype patch(not meant for inclusion) for this io migration

experiment is appended to this e-mail.

Observation #1 mentioned above is also applicable to this experiment. CPU's processing interrupts will now have to cater IO submission/processing load aswell.

Observation #2: This introduces some migration overhead during IO submission. With the current prototype, every incoming IO request results in an IPI and context switch(to kblockd thread) on the interrupt processing cpu. This issue needs to be addressed and main challenge to address is the efficient mechanism of doing this IO migration(how much batching to do and when to send the migrate request?), so that we don't delay the IO much and at the same point, don't cause much overhead during migration.

Source of the IO migration experiment came from an old experiment done in EL3 days (linux-2.4.21-scsi-affine-queue.patch in EL3 GA release, pointed by Arjan). Arjan pointed out that this patch had some perf issues and was taken out in a later update release of EL3. Given that 2.6 has progressed quite a bit from 2.4 days, wondering if we can answer this challenge easily with today's infrastructure.

Experiment - 1 above can be easily incorporated in to linux kernel(by doing the proactive IO cmd completion processing only on cpu processing the interrupts) . We need to address the scheduler load balancing issue (of taking irq load in to account) though.

Is there a simple and better way to efficiently migrate the IO request(perhaps only for direct IO and also based on IO load --- similar to what is pursued in EL3)? Efficient IO migration will further improve the performance numbers stated above.

io migration prototype(and really dirty) patch follows:

```
diff -pNru linux-2.6.21-rc7/block/ll_rw_blk.c linux-batch-delay/block/ll_rw_blk.c
--- linux-2.6.21-rc7/block/ll_rw_blk.c 2007-05-22 18:22:02.000000000 -0700
+++ linux-batch-delay/block/ll_rw_blk.c 2007-06-19 11:56:54.000000000 -0700
@@ -177,6 +177,15 @@ void blk_queue_softirq_done(request_queue
```

```
EXPORT_SYMBOL(blk_queue_softirq_done);
```

```
+static void blk_request_fn_work(struct work_struct *work)
+{
+ request_queue_t *q = container_of(work, request_queue_t, request_fn_work);
+
+ spin_lock_irq(q->queue_lock);
+ q->request_fn(q);
+ spin_unlock_irq(q->queue_lock);
+}
+
+/**
+ * blk_queue_make_request - define an alternate make_request function for a device
```


[rfc] direct IO submission and completion scalability issues

```
clear_bit(Queue_FLAG_REENTER, &q->queue_flags);
} else {
blk_plug_device(q);
- kblockd_schedule_work(&q->unplug_work);
+ if (q->cpu_binding && q->submit_cpu)
+ kblockd_schedule_work_on_cpu(&q->unplug_work, *q->submit_cpu);
+ else
+ kblockd_schedule_work(&q->unplug_work);
}
}

@@ -3627,6 +3649,11 @@ int kblockd_schedule_work(struct work_struct
return queue_work(kblockd_workqueue, work);
}

+int kblockd_schedule_work_on_cpu(struct work_struct *work, int cpu)
+{
+ return queue_work_on_cpu(kblockd_workqueue, work, cpu);
+}
+
EXPORT_SYMBOL(kblockd_schedule_work);

void kblockd_flush(void)
@@ -3813,6 +3840,22 @@ queue_var_store(unsigned long *var, const
return count;
}

+static ssize_t
+queue_cpu_binding_store(struct request_queue *q, const char *page, size_t count)
+{
+ sscanf(page, "%d", &q->cpu_binding);
+ return count;
+}
+
+static ssize_t queue_cpu_binding_show(struct request_queue *q, char *page)
+{
+ int count;
+ count = queue_var_show(q->cpu_binding, (page));
+ if (q->submit_cpu)
+ count += queue_var_show(*q->submit_cpu, (page + count));
+ return count;
+}
+
static ssize_t queue_requests_show(struct request_queue *q, char *page)
{
return queue_var_show(q->nr_requests, (page));
@@ -3946,6 +3989,13 @@ static struct queue_sysfs_entry queue_max
.show = queue_max_hw_sectors_show,
};

+static struct queue_sysfs_entry queue_cpu_binding_entry = {
```

[rfc] direct IO submission and completion scalability issues

```
+ .attr = { .name = "cpu_binding", .mode = S_IRUGO | S_IWUSR },
+ .show = queue_cpu_binding_show,
+ .store = queue_cpu_binding_store,
+};
+
+
static struct queue_sysfs_entry queue_iosched_entry = {
.attr = { .name = "scheduler", .mode = S_IRUGO | S_IWUSR },
.show = elv_iosched_show,
@@ -3958,6 +4008,7 @@ static struct attribute *default_attrs[]
&queue_max_hw_sectors_entry.attr,
&queue_max_sectors_entry.attr,
&queue_iosched_entry.attr,
+ &queue_cpu_binding_entry.attr,
NULL,
};

diff -pNru linux-2.6.21-rc7/drivers/ata/libata-core.c linux-batch-delay/drivers/ata/libata-core.c
--- linux-2.6.21-rc7/drivers/ata/libata-core.c 2007-04-15 16:50:57.000000000 -0700
+++ linux-batch-delay/drivers/ata/libata-core.c 2007-06-19 11:37:29.000000000 -0700
@@ -5223,6 +5223,7 @@ @ @ irqreturn_t ata_interrupt (int irq, void
!(ap->flags & ATA_FLAG_DISABLED)) {
struct ata_queued_cmd *qc;

+ ap->scsi_host->irq_cpu = smp_processor_id();
qc = ata_qc_from_tag(ap, ap->active_tag);
if (qc && (!(qc->tf.flags & ATA_TFLAG_POLLING)) &&
(qc->flags & ATA_QCFLAG_ACTIVE))
diff -pNru linux-2.6.21-rc7/drivers/scsi/qla2xxx/qla_def.h linux-batch-delay/drivers/scsi/qla2xxx/qla_def.h
--- linux-2.6.21-rc7/drivers/scsi/qla2xxx/qla_def.h 2007-04-15 16:50:57.000000000 -0700
+++ linux-batch-delay/drivers/scsi/qla2xxx/qla_def.h 2007-06-19 11:37:29.000000000 -0700
@@ -2294,6 +2294,8 @@ @ @ typedef struct scsi_qla_host {
uint8_t rscn_in_ptr;
uint8_t rscn_out_ptr;

+ unsigned long last_irq_cpu; /* cpu where we got our last irq */
+
/* SNS command interfaces. */
ms_iocb_entry_t *ms_iocb;
dma_addr_t ms_iocb_dma;
diff -pNru linux-2.6.21-rc7/drivers/scsi/qla2xxx/qla_isr.c linux-batch-delay/drivers/scsi/qla2xxx/qla_isr.c
--- linux-2.6.21-rc7/drivers/scsi/qla2xxx/qla_isr.c 2007-04-15 16:50:57.000000000 -0700
+++ linux-batch-delay/drivers/scsi/qla2xxx/qla_isr.c 2007-06-19 11:37:29.000000000 -0700
@@ -44,6 +44,7 @@ @ @ qla2100_intr_handler(int irq, void *dev_
return (IRQ_NONE);
}

+ ha->host->irq_cpu = smp_processor_id();
reg = &ha->iobase->isp;
status = 0;
```

[rfc] direct IO submission and completion scalability issues

```
@@ -121,6 +122,7 @@ qla2300_intr_handler(int irq, void *dev_  
return (IRQ_NONE);  
}
```

```
+ ha->host->irq_cpu = smp_processor_id();  
reg = &ha->iobase->isp;  
status = 0;
```

```
@@ -1437,6 +1439,7 @@ qla24xx_intr_handler(int irq, void *dev_  
return IRQ_NONE;  
}
```

```
+ ha->host->irq_cpu = smp_processor_id();  
reg = &ha->iobase->isp24;  
status = 0;
```

```
diff -pNru linux-2.6.21-rc7/drivers/scsi/scsi_scan.c linux-batch-delay/drivers/scsi/scsi_scan.c  
--- linux-2.6.21-rc7/drivers/scsi/scsi_scan.c 2007-04-15 16:50:57.000000000 -0700  
+++ linux-batch-delay/drivers/scsi/scsi_scan.c 2007-06-19 11:37:37.000000000 -0700  
@@ -280,6 +280,8 @@ static struct scsi_device *scsi_alloc_sd  
}
```

```
sdev->request_queue->queuedata = sdev;  
+ if (sdev->host)  
+ sdev->request_queue->submit_cpu = &sdev->host->irq_cpu;  
scsi_adjust_queue_depth(sdev, 0, sdev->host->cmd_per_lun);
```

```
scsi_sysfs_device_initialize(sdev);  
diff -pNru linux-2.6.21-rc7/include/linux/blkdev.h linux-batch-delay/include/linux/blkdev.h  
--- linux-2.6.21-rc7/include/linux/blkdev.h 2007-05-29 17:02:00.000000000 -0700  
+++ linux-batch-delay/include/linux/blkdev.h 2007-06-19 11:37:29.000000000 -0700  
@@ -392,6 +392,7 @@ struct request_queue  
int unplug_thresh; /* After this many requests */  
unsigned long unplug_delay; /* After this many jiffies */  
struct work_struct unplug_work;  
+ struct work_struct request_fn_work;
```

```
struct backing_dev_info backing_dev_info;
```

```
@@ -400,6 +401,8 @@ struct request_queue  
* ll_rw_blk doesn't touch it.  
*/
```

```
void *queuedata;  
+ int cpu_binding;  
+ int *submit_cpu;
```

```
/*  
* queue needs bounce pages for pages above this limit  
@@ -853,6 +856,7 @@ static inline void put_dev_sector(Sector  
  
struct work_struct;
```

[rfc] direct IO submission and completion scalability issues

```
int kblockd_schedule_work(struct work_struct *work);
+int kblockd_schedule_work_on_cpu(struct work_struct *work, int cpu);
void kblockd_flush(void);

#define MODULE_ALIAS_BLOCKDEV(major,minor) \
diff -pNru linux-2.6.21-rc7/include/linux/workqueue.h linux-batch-delay/include/linux/workqueue.h
--- linux-2.6.21-rc7/include/linux/workqueue.h 2007-05-29 17:02:00.000000000 -0700
+++ linux-batch-delay/include/linux/workqueue.h 2007-06-19 11:37:29.000000000 -0700
@@ -168,6 +168,7 @@ extern struct workqueue_struct *__create
extern void destroy_workqueue(struct workqueue_struct *wq);

extern int FASTCALL(queue_work(struct workqueue_struct *wq, struct work_struct *work));
+extern int FASTCALL(queue_work_on_cpu(struct workqueue_struct *wq, struct work_struct *work, int
cpu));
extern int FASTCALL(queue_delayed_work(struct workqueue_struct *wq, struct delayed_work *work,
unsigned long delay));
extern int queue_delayed_work_on(int cpu, struct workqueue_struct *wq,
struct delayed_work *work, unsigned long delay);
diff -pNru linux-2.6.21-rc7/include/scsi/scsi_host.h linux-batch-delay/include/scsi/scsi_host.h
--- linux-2.6.21-rc7/include/scsi/scsi_host.h 2007-04-15 16:50:57.000000000 -0700
+++ linux-batch-delay/include/scsi/scsi_host.h 2007-06-19 11:37:29.000000000 -0700
@@ -635,6 +635,7 @@ struct Scsi_Host {
unsigned char n_io_port;
unsigned char dma_channel;
unsigned int irq;
+ unsigned int irq_cpu;

enum scsi_host_state shost_state;
diff -pNru linux-2.6.21-rc7/kernel/workqueue.c linux-batch-delay/kernel/workqueue.c
--- linux-2.6.21-rc7/kernel/workqueue.c 2007-06-19 13:13:26.000000000 -0700
+++ linux-batch-delay/kernel/workqueue.c 2007-06-19 11:37:29.000000000 -0700
@@ -218,6 +218,20 @@ int fastcall queue_work(struct workqueue
}
EXPORT_SYMBOL_GPL(queue_work);

+int fastcall queue_work_on_cpu(struct workqueue_struct *wq, struct work_struct *work,
+ int cpu)
+{
+ int ret = 0;
+
+ if (!test_and_set_bit(WORK_STRUCT_PENDING, work_data_bits(work))) {
+ BUG_ON(!list_empty(&work->entry));
+ __queue_work(per_cpu_ptr(wq->cpu_wq, cpu), work);
+ ret = 1;
+ }
+ return ret;
+}
+EXPORT_SYMBOL_GPL(queue_work_on_cpu);
+
void delayed_work_timer_fn(unsigned long __data)
```

[rfc] direct IO submission and completion scalability issues

```
{
struct delayed_work *dwork = (struct delayed_work *)__data;
@@ -351,11 +365,15 @@ static int worker_thread(void *__cwq)
DECLARE_WAITQUEUE(wait, current);
struct k_sigaction sa;
sigset_t blocked;
+ struct sched_param param = { .sched_priority = MAX_RT_PRIO-1 };

if (!cwq->freezeable)
current->flags |= PF_NOFREEZE;

- set_user_nice(current, -5);
+ if (!strncmp(cwq->wq->name, "kblockd", 7))
+ sched_setscheduler(current, SCHED_FIFO, &param);
+ else
+ set_user_nice(current, -5);

/* Block and flush all signals */
sigfillset(&blocked);
-
To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
the body of a message to majordomo@xxxxxxxxxxxxxxxxxxx
More majordomo info at http://vger.kernel.org/majordomo-info.html
Please read the FAQ at http://www.tux.org/lkml/
```