Subject: Subject: [PATCH 1/2] dm-ioband: I/O bandwidth controller v1.3.0: Source code and patch
Posted by Ryo Tsuruta on Fri, 11 Jul 2008 11:15:50 GMT
View Forum Message <> Reply to Message

Here is the patch of dm-ioband.

Based on 2.6.26-rc5-mm3
Signed-off-by: Ryo Tsuruta <ryov@valinux.co.jp>
Signed-off-by: Hirokazu Takahashi <taka@valinux.co.jp>

```
diff -uprN linux-2.6.26-rc5-mm3.orig/drivers/md/Kconfig linux-2.6.26-rc5-mm3/drivers/md/Kconfig
--- linux-2.6.26-rc5-mm3.orig/drivers/md/Kconfig 2008-06-25 15:58:50.000000000 +0900
+++ linux-2.6.26-rc5-mm3/drivers/md/Kconfig 2008-07-11 19:24:05.000000000 +0900
@@ -271,4 +271,17 @@ config DM_UEVENT
  ---help---
  Generate udev events for DM events.

+config DM_IOBAND
+ tristate "I/O bandwidth control (EXPERIMENTAL)"
+ depends on BLK_DEV_DM && EXPERIMENTAL
+ ---help---
+ This device-mapper target allows to define how the
+ available bandwidth of a storage device should be
+ shared between processes, cgroups, the partitions or the LUNs.
+
+ Information on how to use dm-ioband is available in:
+    <file:Documentation/device-mapper/ioband.txt>.
+
+ If unsure, say N.
+
 endif # MD
diff -uprN linux-2.6.26-rc5-mm3.orig/drivers/md/Makefile linux-2.6.26-rc5-mm3/drivers/md/Makefile
--- linux-2.6.26-rc5-mm3.orig/drivers/md/Makefile 2008-06-25 15:58:50.000000000 +0900
+++ linux-2.6.26-rc5-mm3/drivers/md/Makefile 2008-07-11 19:24:05.000000000 +0900
@@ -7,6 +7,7 @@ dm-mod-objs := dm.o dm-table.o dm-target
 dm-multipath-objs := dm-path-selector.o dm-mpath.o
 dm-snapshot-objs := dm-snap.o dm-exception-store.o
 dm-mirror-objs := dm-raid1.o
+dm-ioband-objs := dm-ioband-ctl.o dm-ioband-policy.o dm-ioband-type.o
 md-mod-objs     := md.o bitmap.o
 raid456-objs := raid5.o raid6algos.o raid6recov.o raid6tables.o \
     raid6int1.o raid6int2.o raid6int4.o \
@@ -36,6 +37,7 @@ obj-$(CONFIG_DM_MULTIPATH) += dm-multipa
 obj-$(CONFIG_DM_SNAPSHOT) += dm-snapshot.o
 obj-$(CONFIG_DM_MIRROR)  += dm-mirror.o dm-log.o
 obj-$(CONFIG_DM_ZERO)  += dm-zero.o
+obj-$(CONFIG_DM_IOBAND)  += dm-ioband.o
```

```
 quiet_cmd_unroll = UNROLL  $@
       cmd_unroll = $(PERL) $(srctree)/$(src)/unroll.pl $(UNROLL) \
diff -uprN linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband-ctl.c
linux-2.6.26-rc5-mm3/drivers/md/dm-ioband-ctl.c
--- linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband-ctl.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.26-rc5-mm3/drivers/md/dm-ioband-ctl.c 2008-07-11 19:24:05.000000000 +0900
@@ -0,0 +1,1319 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ * Authors: Hirokazu Takahashi <taka@valinux.co.jp>
+ *          Ryo Tsuruta <ryov@valinux.co.jp>
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+#include <linux/module.h>
+#include <linux/init.h>
+#include <linux/bio.h>
+#include <linux/slab.h>
+#include <linux/workqueue.h>
+#include <linux/raid/md.h>
+#include <linux/rbtree.h>
+#include "dm.h"
+#include "dm-bio-list.h"
+#include "dm-ioband.h"
+
+#define DM_MSG_PREFIX "ioband"
+#define POLICY_PARAM_START 6
+#define POLICY_PARAM_DELIM "=:,"
+
+static LIST_HEAD(ioband_device_list);
+/* to protect ioband_device_list */
+static DEFINE_SPINLOCK(ioband_devicelist_lock);
+
+static void suspend_ioband_device(struct ioband_device *, unsigned long, int);
+static void resume_ioband_device(struct ioband_device *);
+static void ioband_conduct(struct work_struct *);
+static void ioband_hold_bio(struct ioband_group *, struct bio *);
+static struct bio *ioband_pop_bio(struct ioband_group *);
+static int ioband_set_param(struct ioband_group *, char *, char *);
+static int ioband_group_attach(struct ioband_group *, int, char *);
+static int ioband_group_type_select(struct ioband_group *, char *);
+
+long ioband_debug; /* just for debugging */
+
+static void do_nothing(void) {}
```

```
+
+static int policy_init(struct ioband_device *dp, char *name,
+       int argc, char **argv)
+{
+ struct policy_type *p;
+ struct ioband_group *gp;
+ unsigned long flags;
+ int r;
+
+ for (p = dm_ioband_policy_type; p->p_name; p++) {
+  if (!strcmp(name, p->p_name))
+   break;
+ }
+ if (!p->p_name)
+  return -EINVAL;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ if (dp->g_policy == p) {
+  /* do nothing if the same policy is already set */
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  return 0;
+ }
+
+ suspend_ioband_device(dp, flags, 1);
+ list_for_each_entry(gp, &dp->g_groups, c_list)
+  dp->g_group_dtr(gp);
+
+ /* switch to the new policy */
+ dp->g_policy = p;
+ r = p->p_policy_init(dp, argc, argv);
+ if (!dp->g_hold_bio)
+  dp->g_hold_bio = ioband_hold_bio;
+ if (!dp->g_pop_bio)
+  dp->g_pop_bio = ioband_pop_bio;
+
+ list_for_each_entry(gp, &dp->g_groups, c_list)
+  dp->g_group_ctr(gp, NULL);
+ resume_ioband_device(dp);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ return r;
+}
+
+static struct ioband_device *alloc_ioband_device(char *name,
+     int io_throttle, int io_limit)
+
+{
+ struct ioband_device *dp, *new;
+ unsigned long flags;
```

```
+
+ new = kzalloc(sizeof(struct ioband_device), GFP_KERNEL);
+ if (!new)
+  return NULL;
+
+ spin_lock_irqsave(&ioband_devicelist_lock, flags);
+ list_for_each_entry(dp, &ioband_device_list, g_list) {
+  if (!strcmp(dp->g_name, name)) {
+   dp->g_ref++;
+   spin_unlock_irqrestore(&ioband_devicelist_lock, flags);
+   kfree(new);
+   return dp;
+  }
+ }
+
+ /*
+  * Prepare its own workqueue as generic_make_request() may
+  * potentially block the workqueue when submitting BIOs.
+  */
+ new->g_ioband_wq = create_workqueue("kioband");
+ if (!new->g_ioband_wq) {
+  spin_unlock_irqrestore(&ioband_devicelist_lock, flags);
+  kfree(new);
+  return NULL;
+ }
+
+ INIT_DELAYED_WORK(&new->g_conductor, ioband_conduct);
+ INIT_LIST_HEAD(&new->g_groups);
+ INIT_LIST_HEAD(&new->g_list);
+ spin_lock_init(&new->g_lock);
+ mutex_init(&new->g_lock_device);
+ bio_list_init(&new->g_urgent_bios);
+ new->g_io_throttle = io_throttle;
+ new->g_io_limit[0] = io_limit;
+ new->g_io_limit[1] = io_limit;
+ new->g_issued[0] = 0;
+ new->g_issued[1] = 0;
+ new->g_blocked = 0;
+ new->g_ref = 1;
+ new->g_flags = 0;
+ strlcpy(new->g_name, name, sizeof(new->g_name));
+ new->g_policy = NULL;
+ new->g_hold_bio = NULL;
+ new->g_pop_bio = NULL;
+ init_waitqueue_head(&new->g_waitq);
+ init_waitqueue_head(&new->g_waitq_suspend);
+ init_waitqueue_head(&new->g_waitq_flush);
+ list_add_tail(&new->g_list, &ioband_device_list);
```

```
+
+ spin_unlock_irqrestore(&ioband_devicelist_lock, flags);
+ return new;
+}
+
+static void release_ioband_device(struct ioband_device *dp)
+{
+ unsigned long flags;
+
+ spin_lock_irqsave(&ioband_devicelist_lock, flags);
+ dp->g_ref--;
+ if (dp->g_ref > 0) {
+  spin_unlock_irqrestore(&ioband_devicelist_lock, flags);
+  return;
+ }
+ list_del(&dp->g_list);
+ spin_unlock_irqrestore(&ioband_devicelist_lock, flags);
+ destroy_workqueue(dp->g_ioband_wq);
+ kfree(dp);
+}
+
+static int is_ioband_device_flushed(struct ioband_device *dp,
+       int wait_completion)
+{
+ struct ioband_group *gp;
+
+ if (wait_completion && dp->g_issued[0] + dp->g_issued[1] > 0)
+  return 0;
+ if (dp->g_blocked || waitqueue_active(&dp->g_waitq))
+  return 0;
+ list_for_each_entry(gp, &dp->g_groups, c_list)
+  if (waitqueue_active(&gp->c_waitq))
+   return 0;
+ return 1;
+}
+
+static void suspend_ioband_device(struct ioband_device *dp,
+     unsigned long flags, int wait_completion)
+{
+ struct ioband_group *gp;
+
+ /* block incoming bios */
+ set_device_suspended(dp);
+
+ /* wake up all blocked processes and go down all ioband groups */
+ wake_up_all(&dp->g_waitq);
+ list_for_each_entry(gp, &dp->g_groups, c_list) {
+  if (!is_group_down(gp)) {
```

```
+   set_group_down(gp);
+   set_group_need_up(gp);
+  }
+  wake_up_all(&gp->c_waitq);
+ }
+
+ /* flush the already mapped bios */
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ queue_delayed_work(dp->g_ioband_wq, &dp->g_conductor, 0);
+ flush_workqueue(dp->g_ioband_wq);
+
+ /* wait for all processes to wake up and bios to release */
+ spin_lock_irqsave(&dp->g_lock, flags);
+ wait_event_lock_irq(dp->g_waitq_flush,
+   is_ioband_device_flushed(dp, wait_completion),
+   dp->g_lock, do_nothing());
+}
+
+static void resume_ioband_device(struct ioband_device *dp)
+{
+ struct ioband_group *gp;
+
+ /* go up ioband groups */
+ list_for_each_entry(gp, &dp->g_groups, c_list) {
+  if (group_need_up(gp)) {
+   clear_group_need_up(gp);
+   clear_group_down(gp);
+  }
+ }
+
+ /* accept incoming bios */
+ wake_up_all(&dp->g_waitq_suspend);
+ clear_device_suspended(dp);
+}
+
+static struct ioband_group *ioband_group_find(
+     struct ioband_group *head, int id)
+{
+ struct rb_node *node = head->c_group_root.rb_node;
+
+ while (node) {
+  struct ioband_group *p =
+   container_of(node, struct ioband_group, c_group_node);
+
+  if (p->c_id == id || id == IOBAND_ID_ANY)
+   return p;
+  node = (id < p->c_id) ? node->rb_left : node->rb_right;
+ }
```

```
+ return NULL;
+}
+
+static void ioband_group_add_node(struct rb_root *root,
+      struct ioband_group *gp)
+{
+ struct rb_node **new = &root->rb_node, *parent = NULL;
+ struct ioband_group *p;
+
+ while (*new) {
+  p = container_of(*new, struct ioband_group, c_group_node);
+  parent = *new;
+  new = (gp->c_id < p->c_id) ?
+     &(*new)->rb_left : &(*new)->rb_right;
+ }
+
+ rb_link_node(&gp->c_group_node, parent, new);
+ rb_insert_color(&gp->c_group_node, root);
+}
+
+static int ioband_group_init(struct ioband_group *gp,
+    struct ioband_group *head, struct ioband_device *dp, int id, char *param)
+{
+ unsigned long flags;
+ int r;
+
+ INIT_LIST_HEAD(&gp->c_list);
+ bio_list_init(&gp->c_blocked_bios);
+ bio_list_init(&gp->c_prio_bios);
+ gp->c_id = id; /* should be verified */
+ gp->c_blocked = 0;
+ gp->c_prio_blocked = 0;
+ memset(gp->c_stat, 0, sizeof(gp->c_stat));
+ init_waitqueue_head(&gp->c_waitq);
+ gp->c_flags = 0;
+ gp->c_group_root = RB_ROOT;
+ gp->c_banddev = dp;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ if (head && ioband_group_find(head, id)) {
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  DMWARN("ioband_group: id=%d already exists.", id);
+  return -EEXIST;
+ }
+
+ list_add_tail(&gp->c_list, &dp->g_groups);
+
+ r = dp->g_group_ctr(gp, param);
```

```
+ if (r) {
+  list_del(&gp->c_list);
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  return r;
+ }
+
+ if (head) {
+  ioband_group_add_node(&head->c_group_root, gp);
+  gp->c_dev = head->c_dev;
+  gp->c_target = head->c_target;
+ }
+
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ return 0;
+}
+
+static void ioband_group_release(struct ioband_group *head,
+       struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ list_del(&gp->c_list);
+ if (head)
+  rb_erase(&gp->c_group_node, &head->c_group_root);
+ dp->g_group_dtr(gp);
+ kfree(gp);
+}
+
+static void ioband_group_destroy_all(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct ioband_group *group;
+ unsigned long flags;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ while ((group = ioband_group_find(gp, IOBAND_ID_ANY)))
+  ioband_group_release(gp, group);
+ ioband_group_release(NULL, gp);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+}
+
+static void ioband_group_stop_all(struct ioband_group *head, int suspend)
+{
+ struct ioband_device *dp = head->c_banddev;
+ struct ioband_group *p;
+ struct rb_node *node;
+ unsigned long flags;
```

```
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ for (node = rb_first(&head->c_group_root); node; node = rb_next(node)) {
+  p = rb_entry(node, struct ioband_group, c_group_node);
+  set_group_down(p);
+  if (suspend) {
+   set_group_suspended(p);
+   dprintk(KERN_ERR "ioband suspend: gp(%p)\n", p);
+  }
+ }
+ set_group_down(head);
+ if (suspend) {
+  set_group_suspended(head);
+  dprintk(KERN_ERR "ioband suspend: gp(%p)\n", head);
+ }
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ queue_delayed_work(dp->g_ioband_wq, &dp->g_conductor, 0);
+ flush_workqueue(dp->g_ioband_wq);
+}
+
+static void ioband_group_resume_all(struct ioband_group *head)
+{
+ struct ioband_device *dp = head->c_banddev;
+ struct ioband_group *p;
+ struct rb_node *node;
+ unsigned long flags;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ for (node = rb_first(&head->c_group_root); node;
+       node = rb_next(node)) {
+  p = rb_entry(node, struct ioband_group, c_group_node);
+  clear_group_down(p);
+  clear_group_suspended(p);
+  dprintk(KERN_ERR "ioband resume: gp(%p)\n", p);
+ }
+ clear_group_down(head);
+ clear_group_suspended(head);
+ dprintk(KERN_ERR "ioband resume: gp(%p)\n", head);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+}
+
+static int split_string(char *s, long *id, char **v)
+{
+ char *p, *q;
+ int r = 0;
+
+ *id = IOBAND_ID_ANY;
+ p = strsep(&s, POLICY_PARAM_DELIM);
```

```
+ q = strsep(&s, POLICY_PARAM_DELIM);
+ if (!q) {
+  *v = p;
+ } else {
+  r = strict_strtol(p, 0, id);
+  *v = q;
+ }
+ return r;
+}
+
+/*
+ * Create a new band device:
+ *   parameters:  <device> <device-group-id> <io_throttle> <io_limit>
+ *     <type> <policy> <policy-param...> <group-id:group-param...>
+ */
+static int ioband_ctr(struct dm_target *ti, unsigned int argc, char **argv)
+{
+ struct ioband_group *gp;
+ struct ioband_device *dp;
+ struct dm_dev *dev;
+ int io_throttle;
+ int io_limit;
+ int i, r, start;
+ long val, id;
+ char *param;
+
+ if (argc < POLICY_PARAM_START) {
+  ti->error = "Requires " __stringify(POLICY_PARAM_START)
+       " or more arguments";
+  return -EINVAL;
+ }
+
+ if (strlen(argv[1]) > IOBAND_NAME_MAX) {
+  ti->error = "Ioband device name is too long";
+  return -EINVAL;
+ }
+ dprintk(KERN_ERR "ioband_ctr ioband device name:%s\n", argv[1]);
+
+ r = strict_strtol(argv[2], 0, &val);
+ if (r || val < 0) {
+  ti->error = "Invalid io_throttle";
+  return -EINVAL;
+ }
+ io_throttle = (val == 0) ? DEFAULT_IO_THROTTLE : val;
+
+ r = strict_strtol(argv[3], 0, &val);
+ if (r || val < 0) {
+  ti->error = "Invalid io_limit";
```

```
+  return -EINVAL;
+ }
+ io_limit = val;
+
+ r = dm_get_device(ti, argv[0], 0, ti->len,
+    dm_table_get_mode(ti->table), &dev);
+ if (r) {
+  ti->error = "Device lookup failed";
+  return r;
+ }
+
+ if (io_limit == 0) {
+  struct request_queue *q;
+
+  q = bdev_get_queue(dev->bdev);
+  if (!q) {
+   ti->error = "Can't get queue size";
+   r = -ENXIO;
+   goto release_dm_device;
+  }
+  dprintk(KERN_ERR "ioband_ctr nr_requests:%lu\n",
+       q->nr_requests);
+  io_limit = q->nr_requests;
+ }
+
+ if (io_limit < io_throttle)
+  io_limit = io_throttle;
+ dprintk(KERN_ERR "ioband_ctr io_throttle:%d io_limit:%d\n",
+      io_throttle, io_limit);
+
+ dp = alloc_ioband_device(argv[1], io_throttle, io_limit);
+ if (!dp) {
+  ti->error = "Cannot create ioband device";
+  r = -EINVAL;
+  goto release_dm_device;
+ }
+
+ mutex_lock(&dp->g_lock_device);
+ r = policy_init(dp, argv[POLICY_PARAM_START - 1],
+   argc - POLICY_PARAM_START, &argv[POLICY_PARAM_START]);
+ if (r) {
+  ti->error = "Invalid policy parameter";
+  goto release_ioband_device;
+ }
+
+ gp = kzalloc(sizeof(struct ioband_group), GFP_KERNEL);
+ if (!gp) {
+  ti->error = "Cannot allocate memory for ioband group";
```

```
+  r = -ENOMEM;
+  goto release_ioband_device;
+ }
+
+ ti->private = gp;
+ gp->c_target = ti;
+ gp->c_dev = dev;
+
+ /* Find a default group parameter */
+ for (start = POLICY_PARAM_START; start < argc; start++)
+  if (argv[start][0] == ':')
+   break;
+ param = (start < argc) ? &argv[start][1] : NULL;
+
+ /* Create a default ioband group */
+ r = ioband_group_init(gp, NULL, dp, IOBAND_ID_ANY, param);
+ if (r) {
+  kfree(gp);
+  ti->error = "Cannot create default ioband group";
+  goto release_ioband_device;
+ }
+
+ r = ioband_group_type_select(gp, argv[4]);
+ if (r) {
+  ti->error = "Cannot set ioband group type";
+  goto release_ioband_group;
+ }
+
+ /* Create sub ioband groups */
+ for (i = start + 1; i < argc; i++) {
+  r = split_string(argv[i], &id, &param);
+  if (r) {
+   ti->error = "Invalid ioband group parameter";
+   goto release_ioband_group;
+  }
+  r = ioband_group_attach(gp, id, param);
+  if (r) {
+   ti->error = "Cannot create ioband group";
+   goto release_ioband_group;
+  }
+ }
+ mutex_unlock(&dp->g_lock_device);
+ return 0;
+
+release_ioband_group:
+ ioband_group_destroy_all(gp);
+release_ioband_device:
+ mutex_unlock(&dp->g_lock_device);
```

```
+ release_ioband_device(dp);
+release_dm_device:
+ dm_put_device(ti, dev);
+ return r;
+}
+
+static void ioband_dtr(struct dm_target *ti)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+
+ mutex_lock(&dp->g_lock_device);
+ ioband_group_stop_all(gp, 0);
+ cancel_delayed_work_sync(&dp->g_conductor);
+ dm_put_device(ti, gp->c_dev);
+ ioband_group_destroy_all(gp);
+ mutex_unlock(&dp->g_lock_device);
+ release_ioband_device(dp);
+}
+
+static void ioband_hold_bio(struct ioband_group *gp, struct bio *bio)
+{
+ /* Todo: The list should be split into a read list and a write list */
+ bio_list_add(&gp->c_blocked_bios, bio);
+}
+
+static struct bio *ioband_pop_bio(struct ioband_group *gp)
+{
+ return bio_list_pop(&gp->c_blocked_bios);
+}
+
+static int is_urgent_bio(struct bio *bio)
+{
+ struct page *page = bio_iovec_idx(bio, 0)->bv_page;
+ /*
+  * ToDo: A new flag should be added to struct bio, which indicates
+  *  it contains urgent I/O requests.
+  */
+ if (!PageReclaim(page))
+  return 0;
+ if (PageSwapCache(page))
+  return 2;
+ return 1;
+}
+
+static inline void resume_to_accept_bios(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
```

```
+
+ if (is_device_blocked(dp)
+     && dp->g_blocked < dp->g_io_limit[0]+dp->g_io_limit[1]) {
+  clear_device_blocked(dp);
+  wake_up_all(&dp->g_waitq);
+ }
+ if (is_group_blocked(gp)) {
+  clear_group_blocked(gp);
+  wake_up_all(&gp->c_waitq);
+ }
+}
+
+static inline int device_should_block(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (is_group_down(gp))
+  return 0;
+ if (is_device_blocked(dp))
+  return 1;
+ if (dp->g_blocked >= dp->g_io_limit[0] + dp->g_io_limit[1]) {
+  set_device_blocked(dp);
+  return 1;
+ }
+ return 0;
+}
+
+static inline int group_should_block(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (is_group_down(gp))
+  return 0;
+ if (is_group_blocked(gp))
+  return 1;
+ if (dp->g_should_block(gp)) {
+  set_group_blocked(gp);
+  return 1;
+ }
+ return 0;
+}
+
+static void prevent_burst_bios(struct ioband_group *gp, struct bio *bio)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (current->flags & PF_KTHREAD || is_urgent_bio(bio)) {
+  /*
```

```
+	 * Kernel threads shouldn't be blocked easily since each of
+	 * them may handle BIOs for several groups on several
+	 * partitions.
+	 */
+	wait_event_lock_irq(dp->g_waitq, !device_should_block(gp),
+	    dp->g_lock, do_nothing());
+ } else {
+	wait_event_lock_irq(gp->c_waitq, !group_should_block(gp),
+	    dp->g_lock, do_nothing());
+ }
+}
+
+static inline int should_pushback_bio(struct ioband_group *gp)
+{
+ return is_group_suspended(gp) && dm_noflush_suspending(gp->c_target);
+}
+
+static inline int prepare_to_issue(struct ioband_group *gp, struct bio *bio)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ dp->g_issued[bio_data_dir(bio)]++;
+ return dp->g_prepare_bio(gp, bio, 0);
+}
+
+static inline int room_for_bio(struct ioband_device *dp)
+{
+ return dp->g_issued[0] < dp->g_io_limit[0]
+	|| dp->g_issued[1] < dp->g_io_limit[1];
+}
+
+static void hold_bio(struct ioband_group *gp, struct bio *bio)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ dp->g_blocked++;
+ if (is_urgent_bio(bio)) {
+	/*
+	 * ToDo:
+	 * When barrier mode is supported, write bios sharing the same
+	 * file system with the currnt one would be all moved
+	 * to g_urgent_bios list.
+	 * You don't have to care about barrier handling if the bio
+	 * is for swapping.
+	 */
+	dp->g_prepare_bio(gp, bio, IOBAND_URGENT);
+	bio_list_add(&dp->g_urgent_bios, bio);
+ } else {
```

```
+  gp->c_blocked++;
+  dp->g_hold_bio(gp, bio);
+ }
+}
+
+static inline int room_for_bio_rw(struct ioband_device *dp, int direct)
+{
+ return dp->g_issued[direct] < dp->g_io_limit[direct];
+}
+
+static void push_prio_bio(struct ioband_group *gp, struct bio *bio, int direct)
+{
+ if (bio_list_empty(&gp->c_prio_bios))
+  set_prio_queue(gp, direct);
+ bio_list_add(&gp->c_prio_bios, bio);
+ gp->c_prio_blocked++;
+}
+
+static struct bio *pop_prio_bio(struct ioband_group *gp)
+{
+ struct bio *bio = bio_list_pop(&gp->c_prio_bios);
+
+ if (bio_list_empty(&gp->c_prio_bios))
+  clear_prio_queue(gp);
+
+ if (bio)
+  gp->c_prio_blocked--;
+ return bio;
+}
+
+static int make_issue_list(struct ioband_group *gp, struct bio *bio,
+   struct bio_list *issue_list, struct bio_list *pushback_list)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ dp->g_blocked--;
+ gp->c_blocked--;
+ if (!gp->c_blocked)
+  resume_to_accept_bios(gp);
+ if (should_pushback_bio(gp))
+  bio_list_add(pushback_list, bio);
+ else {
+  int rw = bio_data_dir(bio);
+
+  gp->c_stat[rw].deferred++;
+  gp->c_stat[rw].sectors += bio_sectors(bio);
+  bio_list_add(issue_list, bio);
+ }
```

```
+ return prepare_to_issue(gp, bio);
+}
+
+static void release_urgent_bios(struct ioband_device *dp,
+  struct bio_list *issue_list, struct bio_list *pushback_list)
+{
+ struct bio *bio;
+
+ if (bio_list_empty(&dp->g_urgent_bios))
+  return;
+ while (room_for_bio_rw(dp, 1)) {
+  bio = bio_list_pop(&dp->g_urgent_bios);
+  if (!bio)
+   return;
+  dp->g_blocked--;
+  dp->g_issued[1]++;
+  bio_list_add(issue_list, bio);
+ }
+}
+
+static int release_prio_bios(struct ioband_group *gp,
+  struct bio_list *issue_list, struct bio_list *pushback_list)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct bio *bio;
+ int direct;
+ int ret;
+
+ if (bio_list_empty(&gp->c_prio_bios))
+  return R_OK;
+ direct = prio_queue_direct(gp);
+ while (gp->c_prio_blocked) {
+  if (!dp->g_can_submit(gp))
+   return R_BLOCK;
+  if (!room_for_bio_rw(dp, direct))
+   return R_OK;
+  bio = pop_prio_bio(gp);
+  if (!bio)
+   return R_OK;
+  ret = make_issue_list(gp, bio, issue_list, pushback_list);
+  if (ret)
+   return ret;
+ }
+ return R_OK;
+}
+
+static int release_norm_bios(struct ioband_group *gp,
+  struct bio_list *issue_list, struct bio_list *pushback_list)
```

```
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct bio *bio;
+ int direct;
+ int ret;
+
+ while (gp->c_blocked - gp->c_prio_blocked) {
+  if (!dp->g_can_submit(gp))
+   return R_BLOCK;
+  if (!room_for_bio(dp))
+   return R_OK;
+  bio = dp->g_pop_bio(gp);
+  if (!bio)
+   return R_OK;
+
+  direct = bio_data_dir(bio);
+  if (!room_for_bio_rw(dp, direct)) {
+   push_prio_bio(gp, bio, direct);
+   continue;
+  }
+  ret = make_issue_list(gp, bio, issue_list, pushback_list);
+  if (ret)
+   return ret;
+ }
+ return R_OK;
+}
+
+static inline int release_bios(struct ioband_group *gp,
+  struct bio_list *issue_list, struct bio_list *pushback_list)
+{
+ int ret = release_prio_bios(gp, issue_list, pushback_list);
+ if (ret)
+  return ret;
+ return release_norm_bios(gp, issue_list, pushback_list);
+}
+
+static struct ioband_group *ioband_group_get(struct ioband_group *head,
+       struct bio *bio)
+{
+ struct ioband_group *gp;
+
+ if (!head->c_type->t_getid)
+  return head;
+
+ gp = ioband_group_find(head, head->c_type->t_getid(bio));
+
+ if (!gp)
+  gp = head;
```

```
+ return gp;
+}
+
+/*
+ * Start to control the bandwidth once the number of uncompleted BIOs
+ * exceeds the value of "io_throttle".
+ */
+static int ioband_map(struct dm_target *ti, struct bio *bio,
+      union map_info *map_context)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+ unsigned long flags;
+ int rw;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+
+ /*
+  * The device is suspended while some of the ioband device
+  * configurations are being changed.
+  */
+ if (is_device_suspended(dp))
+  wait_event_lock_irq(dp->g_waitq_suspend,
+   !is_device_suspended(dp), dp->g_lock, do_nothing());
+
+ gp = ioband_group_get(gp, bio);
+ prevent_burst_bios(gp, bio);
+ if (should_pushback_bio(gp)) {
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  return DM_MAPIO_REQUEUE;
+ }
+
+ bio->bi_bdev = gp->c_dev->bdev;
+ bio->bi_sector -= ti->begin;
+ rw = bio_data_dir(bio);
+
+ if (!gp->c_blocked && room_for_bio_rw(dp, rw)) {
+  if (dp->g_can_submit(gp)) {
+   prepare_to_issue(gp, bio);
+   gp->c_stat[rw].immediate++;
+   gp->c_stat[rw].sectors += bio_sectors(bio);
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   return DM_MAPIO_REMAPPED;
+  } else if (!dp->g_blocked
+    && dp->g_issued[0] + dp->g_issued[1] == 0) {
+   dprintk(KERN_ERR "ioband_map: token expired "
+     "gp:%p bio:%p\n", gp, bio);
+   queue_delayed_work(dp->g_ioband_wq,
```

```
+       &dp->g_conductor, 1);
+  }
+ }
+ hold_bio(gp, bio);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ return DM_MAPIO_SUBMITTED;
+}
+
+/*
+ * Select the best group to resubmit its BIOs.
+ */
+static struct ioband_group *choose_best_group(struct ioband_device *dp)
+{
+ struct ioband_group *gp;
+ struct ioband_group *best = NULL;
+ int highest = 0;
+ int pri;
+
+ /* Todo: The algorithm should be optimized.
+  *       It would be better to use rbtree.
+  */
+ list_for_each_entry(gp, &dp->g_groups, c_list) {
+  if (!gp->c_blocked || !room_for_bio(dp))
+   continue;
+  if (gp->c_blocked == gp->c_prio_blocked
+   && !room_for_bio_rw(dp, prio_queue_direct(gp))) {
+   continue;
+  }
+  pri = dp->g_can_submit(gp);
+  if (pri > highest) {
+   highest = pri;
+   best = gp;
+  }
+ }
+
+ return best;
+}
+
+/*
+ * This function is called right after it becomes able to resubmit BIOs.
+ * It selects the best BIOs and passes them to the underlying layer.
+ */
+static void ioband_conduct(struct work_struct *work)
+{
+ struct ioband_device *dp =
+  container_of(work, struct ioband_device, g_conductor.work);
+ struct ioband_group *gp = NULL;
```

```
+ struct bio *bio;
+ unsigned long flags;
+ struct bio_list issue_list, pushback_list;
+
+ bio_list_init(&issue_list);
+ bio_list_init(&pushback_list);
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ release_urgent_bios(dp, &issue_list, &pushback_list);
+ if (dp->g_blocked) {
+  gp = choose_best_group(dp);
+  if (gp && release_bios(gp, &issue_list, &pushback_list)
+        == R_YIELD)
+   queue_delayed_work(dp->g_ioband_wq,
+       &dp->g_conductor, 0);
+ }
+
+ if (dp->g_blocked && room_for_bio_rw(dp, 0) && room_for_bio_rw(dp, 1) &&
+  bio_list_empty(&issue_list) && bio_list_empty(&pushback_list) &&
+  dp->g_restart_bios(dp)) {
+  dprintk(KERN_ERR "ioband_conduct: token expired dp:%p "
+   "issued(%d,%d) g_blocked(%d)\n", dp,
+    dp->g_issued[0], dp->g_issued[1], dp->g_blocked);
+  queue_delayed_work(dp->g_ioband_wq, &dp->g_conductor, 0);
+ }
+
+
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ while ((bio = bio_list_pop(&issue_list)))
+  generic_make_request(bio);
+ while ((bio = bio_list_pop(&pushback_list)))
+  bio_endio(bio, -EIO);
+}
+
+static int ioband_end_io(struct dm_target *ti, struct bio *bio,
+    int error, union map_info *map_context)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+ unsigned long flags;
+ int r = error;
+
+ /*
+  *  XXX: A new error code for device mapper devices should be used
+  *       rather than EIO.
+  */
+ if (error == -EIO && should_pushback_bio(gp)) {
```

```
+  /* This ioband device is suspending */
+  r = DM_ENDIO_REQUEUE;
+ }
+ /*
+  * Todo: The algorithm should be optimized to eliminate the spinlock.
+  */
+ spin_lock_irqsave(&dp->g_lock, flags);
+ dp->g_issued[bio_data_dir(bio)]--;
+
+ /*
+  * Todo: It would be better to introduce high/low water marks here
+  *    not to kick the workqueues so often.
+  */
+ if (dp->g_blocked)
+  queue_delayed_work(dp->g_ioband_wq, &dp->g_conductor, 0);
+ else if (is_device_suspended(dp)
+     && dp->g_issued[0] + dp->g_issued[1] == 0)
+  wake_up_all(&dp->g_waitq_flush);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ return r;
+}
+
+static void ioband_presuspend(struct dm_target *ti)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+
+ mutex_lock(&dp->g_lock_device);
+ ioband_group_stop_all(gp, 1);
+ mutex_unlock(&dp->g_lock_device);
+}
+
+static void ioband_resume(struct dm_target *ti)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+
+ mutex_lock(&dp->g_lock_device);
+ ioband_group_resume_all(gp);
+ mutex_unlock(&dp->g_lock_device);
+}
+
+
+static void ioband_group_status(struct ioband_group *gp, int *szp,
+     char *result, unsigned int maxlen)
+{
+ struct ioband_group_stat *stat;
+ int i, sz = *szp; /* used in DMEMIT() */
```

```
+
+ DMEMIT(" %d", gp->c_id);
+ for (i = 0; i < 2; i++) {
+  stat = &gp->c_stat[i];
+  DMEMIT(" %lu %lu %lu",
+   stat->immediate + stat->deferred, stat->deferred,
+   stat->sectors);
+ }
+ *szp = sz;
+}
+
+static int ioband_status(struct dm_target *ti, status_type_t type,
+     char *result, unsigned int maxlen)
+{
+ struct ioband_group *gp = ti->private, *p;
+ struct ioband_device *dp = gp->c_banddev;
+ struct rb_node *node;
+ int sz = 0; /* used in DMEMIT() */
+ unsigned long flags;
+
+ mutex_lock(&dp->g_lock_device);
+
+ switch (type) {
+ case STATUSTYPE_INFO:
+  spin_lock_irqsave(&dp->g_lock, flags);
+  DMEMIT("%s", dp->g_name);
+  ioband_group_status(gp, &sz, result, maxlen);
+  for (node = rb_first(&gp->c_group_root); node;
+       node = rb_next(node)) {
+   p = rb_entry(node, struct ioband_group, c_group_node);
+   ioband_group_status(p, &sz, result, maxlen);
+  }
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  break;
+
+ case STATUSTYPE_TABLE:
+  spin_lock_irqsave(&dp->g_lock, flags);
+  DMEMIT("%s %s %d %d %s %s",
+    gp->c_dev->name, dp->g_name,
+    dp->g_io_throttle, dp->g_io_limit[0],
+    gp->c_type->t_name, dp->g_policy->p_name);
+  dp->g_show(gp, &sz, result, maxlen);
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  break;
+ }
+
+ mutex_unlock(&dp->g_lock_device);
+ return 0;
```

```
+}
+
+static int ioband_group_type_select(struct ioband_group *gp, char *name)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct group_type *t;
+ unsigned long flags;
+
+ for (t = dm_ioband_group_type; (t->t_name); t++) {
+  if (!strcmp(name, t->t_name))
+   break;
+ }
+ if (!t->t_name) {
+  DMWARN("ioband type select: %s isn't supported.", name);
+  return -EINVAL;
+ }
+ spin_lock_irqsave(&dp->g_lock, flags);
+ if (!RB_EMPTY_ROOT(&gp->c_group_root)) {
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  return -EBUSY;
+ }
+ gp->c_type = t;
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ return 0;
+}
+
+static int ioband_set_param(struct ioband_group *gp, char *cmd, char *value)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ char *val_str;
+ long id;
+ unsigned long flags;
+ int r;
+
+ r = split_string(value, &id, &val_str);
+ if (r)
+  return r;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ if (id != IOBAND_ID_ANY) {
+  gp = ioband_group_find(gp, id);
+  if (!gp) {
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   DMWARN("ioband_set_param: id=%ld not found.", id);
+   return -EINVAL;
+  }
+ }
```

```
+ r = dp->g_set_param(gp, cmd, val_str);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ return r;
+}
+
+static int ioband_group_attach(struct ioband_group *gp, int id, char *param)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct ioband_group *sub_gp;
+ int r;
+
+ if (id < 0) {
+  DMWARN("ioband_group_attach: invalid id:%d", id);
+  return -EINVAL;
+ }
+ if (!gp->c_type->t_getid) {
+  DMWARN("ioband_group_attach: "
+         "no ioband group type is specified");
+  return -EINVAL;
+ }
+
+ sub_gp = kzalloc(sizeof(struct ioband_group), GFP_KERNEL);
+ if (!sub_gp)
+  return -ENOMEM;
+
+ r = ioband_group_init(sub_gp, gp, dp, id, param);
+ if (r < 0) {
+  kfree(sub_gp);
+  return r;
+ }
+ return 0;
+}
+
+static int ioband_group_detach(struct ioband_group *gp, int id)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct ioband_group *sub_gp;
+ unsigned long flags;
+
+ if (id < 0) {
+  DMWARN("ioband_group_detach: invalid id:%d", id);
+  return -EINVAL;
+ }
+ spin_lock_irqsave(&dp->g_lock, flags);
+ sub_gp = ioband_group_find(gp, id);
+ if (!sub_gp) {
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  DMWARN("ioband_group_detach: invalid id:%d", id);
```

```
+  return -EINVAL;
+ }
+
+ /*
+  * Todo: Calling suspend_ioband_device() before releasing the
+  *       ioband group has a large overhead. Need improvement.
+  */
+ suspend_ioband_device(dp, flags, 0);
+ ioband_group_release(gp, sub_gp);
+ resume_ioband_device(dp);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ return 0;
+}
+
+/*
+ * Message parameters:
+ * "policy"      <name>
+ *       ex)
+ *  "policy" "weight"
+ * "type"        "none"|"pid"|"pgrp"|"node"|"cpuset"|"cgroup"|"user"|"gid"
+ *  "io_throttle" <value>
+ *  "io_limit"    <value>
+ * "attach"      <group id>
+ * "detach"      <group id>
+ * "any-command" <group id>:<value>
+ *       ex)
+ *  "weight" 0:<value>
+ *  "token"  24:<value>
+ */
+static int __ioband_message(struct dm_target *ti,
+     unsigned int argc, char **argv)
+{
+ struct ioband_group *gp = ti->private, *p;
+ struct ioband_device *dp = gp->c_banddev;
+ struct rb_node *node;
+ long val;
+ int r = 0;
+ unsigned long flags;
+
+ if (argc == 1 && !strcmp(argv[0], "reset")) {
+  spin_lock_irqsave(&dp->g_lock, flags);
+  memset(gp->c_stat, 0, sizeof(gp->c_stat));
+  for (node = rb_first(&gp->c_group_root); node;
+       node = rb_next(node)) {
+   p = rb_entry(node, struct ioband_group, c_group_node);
+   memset(p->c_stat, 0, sizeof(p->c_stat));
+  }
+  spin_unlock_irqrestore(&dp->g_lock, flags);
```

```
+  return 0;
+ }
+
+ if (argc != 2) {
+  DMWARN("Unrecognised band message received.");
+  return -EINVAL;
+ }
+ if (!strcmp(argv[0], "debug")) {
+  r = strict_strtol(argv[1], 0, &val);
+  if (r || val < 0)
+   return -EINVAL;
+  ioband_debug = val;
+  return 0;
+ } else if (!strcmp(argv[0], "io_throttle")) {
+  r = strict_strtol(argv[1], 0, &val);
+  spin_lock_irqsave(&dp->g_lock, flags);
+  if (r || val < 0 ||
+   val > dp->g_io_limit[0] || val > dp->g_io_limit[1]) {
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   return -EINVAL;
+  }
+  dp->g_io_throttle = (val == 0) ? DEFAULT_IO_THROTTLE : val;
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  ioband_set_param(gp, argv[0], argv[1]);
+  return 0;
+ } else if (!strcmp(argv[0], "io_limit")) {
+  r = strict_strtol(argv[1], 0, &val);
+  if (r || val < 0)
+   return -EINVAL;
+  spin_lock_irqsave(&dp->g_lock, flags);
+  if (val == 0) {
+   struct request_queue *q;
+
+   q = bdev_get_queue(gp->c_dev->bdev);
+   if (!q) {
+    spin_unlock_irqrestore(&dp->g_lock, flags);
+    return -ENXIO;
+   }
+   val = q->nr_requests;
+  }
+  if (val < dp->g_io_throttle) {
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   return -EINVAL;
+  }
+  dp->g_io_limit[0] = dp->g_io_limit[1] = val;
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  ioband_set_param(gp, argv[0], argv[1]);
+  return 0;
```

```
+ } else if (!strcmp(argv[0], "type")) {
+  return ioband_group_type_select(gp, argv[1]);
+ } else if (!strcmp(argv[0], "attach")) {
+  r = strict_strtol(argv[1], 0, &val);
+  if (r)
+   return r;
+  return ioband_group_attach(gp, val, NULL);
+ } else if (!strcmp(argv[0], "detach")) {
+  r = strict_strtol(argv[1], 0, &val);
+  if (r)
+   return r;
+  return ioband_group_detach(gp, val);
+ } else if (!strcmp(argv[0], "policy")) {
+  r = policy_init(dp, argv[1], 0, &argv[2]);
+  return r;
+ } else {
+  /* message anycommand <group-id>:<value> */
+  r = ioband_set_param(gp, argv[0], argv[1]);
+  if (r < 0)
+   DMWARN("Unrecognised band message received.");
+  return r;
+ }
+ return 0;
+}
+
+static int ioband_message(struct dm_target *ti, unsigned int argc, char **argv)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+ int r;
+
+ mutex_lock(&dp->g_lock_device);
+ r = __ioband_message(ti, argc, argv);
+ mutex_unlock(&dp->g_lock_device);
+ return r;
+}
+
+static struct target_type ioband_target = {
+ .name      = "ioband",
+ .module      = THIS_MODULE,
+ .version     = {1, 3, 0},
+ .ctr      = ioband_ctr,
+ .dtr      = ioband_dtr,
+ .map      = ioband_map,
+ .end_io      = ioband_end_io,
+ .presuspend  = ioband_presuspend,
+ .resume      = ioband_resume,
+ .status      = ioband_status,
```

```
+ .message     = ioband_message,
+};
+
+static int __init dm_ioband_init(void)
+{
+ int r;
+
+ r = dm_register_target(&ioband_target);
+ if (r < 0) {
+  DMERR("register failed %d", r);
+  return r;
+ }
+ return r;
+}
+
+static void __exit dm_ioband_exit(void)
+{
+ int r;
+
+ r = dm_unregister_target(&ioband_target);
+ if (r < 0)
+  DMERR("unregister failed %d", r);
+}
+
+module_init(dm_ioband_init);
+module_exit(dm_ioband_exit);
+
+MODULE_DESCRIPTION(DM_NAME " I/O bandwidth control");
+MODULE_AUTHOR("Hirokazu Takahashi <taka@valinux.co.jp>, "
+       "Ryo Tsuruta <ryov@valinux.co.jp");
+MODULE_LICENSE("GPL");
diff -uprN linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband-policy.c
linux-2.6.26-rc5-mm3/drivers/md/dm-ioband-policy.c
--- linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband-policy.c 1970-01-01 09:00:00.000000000
+0900
+++ linux-2.6.26-rc5-mm3/drivers/md/dm-ioband-policy.c 2008-07-11 19:24:05.000000000 +0900
@@ -0,0 +1,446 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+#include <linux/bio.h>
+#include <linux/workqueue.h>
+#include <linux/rbtree.h>
+#include "dm.h"
```

```
+#include "dm-bio-list.h"
+#include "dm-ioband.h"
+
+/*
+ * The following functions determine when and which BIOs should
+ * be submitted to control the I/O flow.
+ * It is possible to add a new BIO scheduling policy with it.
+ */
+
+
+/*
+ * Functions for weight balancing policy based on the number of I/Os.
+ */
+#define DEFAULT_WEIGHT  100
+#define DEFAULT_TOKENPOOL 2048
+#define DEFAULT_BUCKET  2
+#define IOBAND_IOPRIO_BASE 100
+#define TOKEN_BATCH_UNIT 20
+#define PROCEED_THRESHOLD 8
+#define LOCAL_ACTIVE_RATIO 8
+#define GLOBAL_ACTIVE_RATIO 16
+#define OVERCOMMIT_RATE  4
+
+/*
+ * Calculate the effective number of tokens this group has.
+ */
+static int get_token(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ int token = gp->c_token;
+ int allowance = dp->g_epoch - gp->c_my_epoch;
+
+ if (allowance) {
+  if (allowance > dp->g_carryover)
+   allowance = dp->g_carryover;
+  token += gp->c_token_initial * allowance;
+ }
+ if (is_group_down(gp))
+  token += gp->c_token_initial * dp->g_carryover * 2;
+
+ return token;
+}
+
+/*
+ * Calculate the priority of a given group.
+ */
+static int iopriority(struct ioband_group *gp)
+{
```

```
+ return get_token(gp) * IOBAND_IOPRIO_BASE / gp->c_token_initial + 1;
+}
+
+/*
+ * This function is called when all the active group on the same ioband
+ * device has used up their tokens. It makes a new global epoch so that
+ * all groups on this device will get freshly assigned tokens.
+ */
+static int make_global_epoch(struct ioband_device *dp)
+{
+ struct ioband_group *gp = dp->g_dominant;
+
+ /*
+  * Don't make a new epoch if the dominant group still has a lot of
+  * tokens, except when the I/O load is low.
+  */
+ if (gp) {
+  int iopri = iopriority(gp);
+  if (iopri * PROCEED_THRESHOLD > IOBAND_IOPRIO_BASE &&
+   dp->g_issued[0] + dp->g_issued[1] >= dp->g_io_throttle)
+  return 0;
+ }
+
+ dp->g_epoch++;
+ dprintk(KERN_ERR "make_epoch %d --> %d\n",
+       dp->g_epoch-1, dp->g_epoch);
+
+ /* The leftover tokens will be used in the next epoch. */
+ dp->g_token_extra = dp->g_token_left;
+ if (dp->g_token_extra < 0)
+  dp->g_token_extra = 0;
+ dp->g_token_left = dp->g_token_bucket;
+
+ dp->g_expired = NULL;
+ dp->g_dominant = NULL;
+
+ return 1;
+}
+
+/*
+ * This function is called when this group has used up its own tokens.
+ * It will check whether it's possible to make a new epoch of this group.
+ */
+static inline int make_epoch(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ int allowance = dp->g_epoch - gp->c_my_epoch;
+
```

```
+ if (!allowance)
+  return 0;
+ if (allowance > dp->g_carryover)
+  allowance = dp->g_carryover;
+ gp->c_my_epoch = dp->g_epoch;
+ return allowance;
+}
+
+/*
+ * Check whether this group has tokens to issue an I/O. Return 0 if it
+ * doesn't have any, otherwise return the priority of this group.
+ */
+static int is_token_left(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ int allowance;
+ int delta;
+ int extra;
+
+ if (gp->c_token > 0)
+  return iopriority(gp);
+
+ if (is_group_down(gp)) {
+  gp->c_token = gp->c_token_initial;
+  return iopriority(gp);
+ }
+ allowance = make_epoch(gp);
+ if (!allowance)
+  return 0;
+ /*
+  * If this group has the right to get tokens for several epochs,
+  * give all of them to the group here.
+  */
+ delta = gp->c_token_initial * allowance;
+ dp->g_token_left -= delta;
+ /*
+  * Give some extra tokens to this group when there have left unused
+  * tokens on this ioband device from the previous epoch.
+  */
+ extra = dp->g_token_extra * gp->c_token_initial /
+     (dp->g_token_bucket - dp->g_token_extra/2);
+ delta += extra;
+ gp->c_token += delta;
+ gp->c_consumed = 0;
+
+ if (gp == dp->g_current)
+  dp->g_yield_mark += delta;
+ dprintk(KERN_ERR "refill token: "
```

```
+  "gp:%p token:%d->%d extra(%d) allowance(%d)\n",
+  gp, gp->c_token - delta, gp->c_token, extra, allowance);
+ if (gp->c_token > 0)
+  return iopriority(gp);
+ dprintk(KERN_ERR "refill token: yet empty gp:%p token:%d\n",
+      gp, gp->c_token);
+ return 0;
+}
+
+/*
+ * Use tokens to issue an I/O. After the operation, the number of tokens left
+ * on this group may become negative value, which will be treated as debt.
+ */
+static int consume_token(struct ioband_group *gp, int count, int flag)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (gp->c_consumed * LOCAL_ACTIVE_RATIO < gp->c_token_initial &&
+  gp->c_consumed * GLOBAL_ACTIVE_RATIO < dp->g_token_bucket) {
+  ; /* Do nothing unless this group is really active. */
+ } else if (!dp->g_dominant ||
+   get_token(gp) > get_token(dp->g_dominant)) {
+  /*
+   * Regard this group as the dominant group on this
+   * ioband device when it has larger number of tokens
+   * than those of the previous one.
+   */
+  dp->g_dominant = gp;
+ }
+ if (dp->g_epoch == gp->c_my_epoch &&
+   gp->c_token > 0 && gp->c_token - count <= 0) {
+  /* Remember the last group which used up its own tokens. */
+  dp->g_expired = gp;
+  if (dp->g_dominant == gp)
+   dp->g_dominant = NULL;
+ }
+
+ if (gp != dp->g_current) {
+  /* This group is the current already. */
+  dp->g_current = gp;
+  dp->g_yield_mark =
+   gp->c_token - (TOKEN_BATCH_UNIT << dp->g_token_unit);
+ }
+ gp->c_token -= count;
+ gp->c_consumed += count;
+ if (gp->c_token <= dp->g_yield_mark && !(flag & IOBAND_URGENT)) {
+  /*
+   * Return-value 1 means that this policy requests dm-ioband
```

```
+   * to give a chance to another group to be selected since
+   * this group has already issued enough amount of I/Os.
+   */
+  dp->g_current = NULL;
+  return R_YIELD;
+ }
+ /*
+  * Return-value 0 means that this policy allows dm-ioband to select
+  * this group to issue I/Os without a break.
+  */
+ return R_OK;
+}
+
+/*
+ * Consume one token on each I/O.
+ */
+static int prepare_token(struct ioband_group *gp, struct bio *bio, int flag)
+{
+ return consume_token(gp, 1, flag);
+}
+
+/*
+ * Check if this group is able to receive a new bio.
+ */
+static int is_queue_full(struct ioband_group *gp)
+{
+ return gp->c_blocked >= gp->c_limit;
+}
+
+static void set_weight(struct ioband_group *gp, int new)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ struct ioband_group *p;
+
+ dp->g_weight_total += (new - gp->c_weight);
+ gp->c_weight = new;
+
+ if (dp->g_weight_total == 0) {
+  list_for_each_entry(p, &dp->g_groups, c_list)
+   p->c_token = p->c_token_initial = p->c_limit = 1;
+ } else {
+  list_for_each_entry(p, &dp->g_groups, c_list) {
+   p->c_token = p->c_token_initial =
+    dp->g_token_bucket * p->c_weight /
+    dp->g_weight_total + 1;
+   p->c_limit = (dp->g_io_limit[0] + dp->g_io_limit[1]) *
+    p->c_weight / dp->g_weight_total /
+    OVERCOMMIT_RATE + 1;
```

```
+  }
+ }
+}
+
+static void init_token_bucket(struct ioband_device *dp, int val)
+{
+ dp->g_token_bucket = ((dp->g_io_limit[0] + dp->g_io_limit[1]) *
+     DEFAULT_BUCKET) << dp->g_token_unit;
+ if (!val)
+  val = DEFAULT_TOKENPOOL << dp->g_token_unit;
+ else if (val < dp->g_token_bucket)
+  val = dp->g_token_bucket;
+ dp->g_carryover = val/dp->g_token_bucket;
+}
+
+static int policy_weight_param(struct ioband_group *gp, char *cmd, char *value)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ long val;
+ int r = 0, err;
+
+ err = strict_strtol(value, 0, &val);
+ if (!strcmp(cmd, "weight")) {
+  if (!err && 0 < val && val <= SHORT_MAX)
+   set_weight(gp, val);
+  else
+   r = -EINVAL;
+ } else if (!strcmp(cmd, "token")) {
+  if (!err && val > 0) {
+   init_token_bucket(dp, val);
+   set_weight(gp, gp->c_weight);
+   dp->g_token_extra = 0;
+  } else
+   r = -EINVAL;
+ } else if (!strcmp(cmd, "io_limit")) {
+  init_token_bucket(dp, dp->g_token_bucket * dp->g_carryover);
+  set_weight(gp, gp->c_weight);
+ } else {
+  r = -EINVAL;
+ }
+ return r;
+}
+
+static int policy_weight_ctr(struct ioband_group *gp, char *arg)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (!arg)
```

```
+  arg = __stringify(DEFAULT_WEIGHT);
+ gp->c_my_epoch = dp->g_epoch;
+ gp->c_weight = 0;
+ gp->c_consumed = 0;
+ return policy_weight_param(gp, "weight", arg);
+}
+
+static void policy_weight_dtr(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ set_weight(gp, 0);
+ dp->g_dominant = NULL;
+ dp->g_expired = NULL;
+}
+
+static void policy_weight_show(struct ioband_group *gp, int *szp,
+     char *result, unsigned int maxlen)
+{
+ struct ioband_group *p;
+ struct ioband_device *dp = gp->c_banddev;
+ struct rb_node *node;
+ int sz = *szp; /* used in DMEMIT() */
+
+ DMEMIT(" %d :%d", dp->g_token_bucket * dp->g_carryover, gp->c_weight);
+
+ for (node = rb_first(&gp->c_group_root); node; node = rb_next(node)) {
+  p = rb_entry(node, struct ioband_group, c_group_node);
+  DMEMIT(" %d:%d", p->c_id, p->c_weight);
+ }
+ *szp = sz;
+}
+
+/*
+ *  <Method>      <description>
+ * g_can_submit   : To determine whether a given group has the right to
+ *                  submit BIOs. The larger the return value the higher the
+ *                  priority to submit. Zero means it has no right.
+ * g_prepare_bio  : Called right before submitting each BIO.
+ * g_restart_bios : Called if this ioband device has some BIOs blocked but none
+ *                  of them can be submitted now. This method has to
+ *                  reinitialize the data to restart to submit BIOs and return
+ *                  0 or 1.
+ *                  The return value 0 means that it has become able to submit
+ *                  them now so that this ioband device will continue its work.
+ *                  The return value 1 means that it is still unable to submit
+ *                  them so that this device will stop its work. And this
+ *                  policy module has to reactivate the device when it gets
+ *                  to be able to submit BIOs.
```

```
+ * g_hold_bio     : To hold a given BIO until it is submitted.
+ *                 The default function is used when this method is undefined.
+ * g_pop_bio      : To select and get the best BIO to submit.
+ * g_group_ctr    : To initalize the policy own members of struct ioband_group.
+ * g_group_dtr    : Called when struct ioband_group is removed.
+ * g_set_param    : To update the policy own date.
+ *                 The parameters can be passed through "dmsetup message"
+ *                 command.
+ * g_should_block : Called every time this ioband device receive a BIO.
+ *                 Return 1 if a given group can't receive any more BIOs,
+ *                 otherwise return 0.
+ * g_show         : Show the configuration.
+ */
+static int policy_weight_init(struct ioband_device *dp, int argc, char **argv)
+{
+ long val;
+ int r = 0;
+
+ if (argc < 1)
+  val = 0;
+ else {
+  r = strict_strtol(argv[0], 0, &val);
+  if (r || val < 0)
+   return -EINVAL;
+ }
+
+ dp->g_can_submit = is_token_left;
+ dp->g_prepare_bio = prepare_token;
+ dp->g_restart_bios = make_global_epoch;
+ dp->g_group_ctr = policy_weight_ctr;
+ dp->g_group_dtr = policy_weight_dtr;
+ dp->g_set_param = policy_weight_param;
+ dp->g_should_block = is_queue_full;
+ dp->g_show  = policy_weight_show;
+
+ dp->g_epoch = 0;
+ dp->g_weight_total = 0;
+ dp->g_current = NULL;
+ dp->g_dominant = NULL;
+ dp->g_expired = NULL;
+ dp->g_token_extra = 0;
+ dp->g_token_unit = 0;
+ init_token_bucket(dp, val);
+ dp->g_token_left = dp->g_token_bucket;
+
+ return 0;
+}
+/* weight balancing policy based on the number of I/Os. --- End --- */
```

```
+
+
+/*
+ * Functions for weight balancing policy based on I/O size.
+ * It just borrows a lot of functions from the regular weight balancing policy.
+ */
+static int w2_prepare_token(struct ioband_group *gp, struct bio *bio, int flag)
+{
+ /* Consume tokens depending on the size of a given bio. */
+ return consume_token(gp, bio_sectors(bio), flag);
+}
+
+static int w2_policy_weight_init(struct ioband_device *dp,
+       int argc, char **argv)
+{
+ long val;
+ int r = 0;
+
+ if (argc < 1)
+  val = 0;
+ else {
+  r = strict_strtol(argv[0], 0, &val);
+  if (r || val < 0)
+   return -EINVAL;
+ }
+
+ r = policy_weight_init(dp, argc, argv);
+ if (r < 0)
+  return r;
+
+ dp->g_prepare_bio = w2_prepare_token;
+ dp->g_token_unit = PAGE_SHIFT - 9;
+ init_token_bucket(dp, val);
+ dp->g_token_left = dp->g_token_bucket;
+ return 0;
+}
+/* weight balancing policy based on I/O size. --- End --- */
+
+
+static int policy_default_init(struct ioband_device *dp,
+     int argc, char **argv)
+{
+ return policy_weight_init(dp, argc, argv);
+}
+
+struct policy_type dm_ioband_policy_type[] = {
+ {"default", policy_default_init},
+ {"weight", policy_weight_init},
```

```
+ {"weight-iosize", w2_policy_weight_init},
+ {NULL,     policy_default_init}
+};
diff -uprN linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband-type.c
linux-2.6.26-rc5-mm3/drivers/md/dm-ioband-type.c
--- linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband-type.c 1970-01-01 09:00:00.000000000
+0900
+++ linux-2.6.26-rc5-mm3/drivers/md/dm-ioband-type.c 2008-07-11 19:24:05.000000000 +0900
@@ -0,0 +1,76 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
+ */
+#include <linux/bio.h>
+#include "dm.h"
+#include "dm-bio-list.h"
+#include "dm-ioband.h"
+
+/*
+ * Any I/O bandwidth can be divided into several bandwidth groups, each of which
+ * has its own unique ID. The following functions are called to determine
+ * which group a given BIO belongs to and return the ID of the group.
+ */
+
+/* ToDo: unsigned long value would be better for group ID */
+
+static int ioband_process_id(struct bio *bio)
+{
+ /*
+  * This function will work for KVM and Xen.
+  */
+ return (int)current->tgid;
+}
+
+static int ioband_process_group(struct bio *bio)
+{
+ return (int)task_pgrp_nr(current);
+}
+
+static int ioband_uid(struct bio *bio)
+{
+ return (int)current->uid;
+}
+
+static int ioband_gid(struct bio *bio)
```

```
+{
+ return (int)current->gid;
+}
+
+static int ioband_cpuset(struct bio *bio)
+{
+ return 0; /* not implemented yet */
+}
+
+static int ioband_node(struct bio *bio)
+{
+ return 0; /* not implemented yet */
+}
+
+static int ioband_cgroup(struct bio *bio)
+{
+  /*
+   * This function should return the ID of the cgroup which issued "bio".
+   * The ID of the cgroup which the current process belongs to won't be
+   * suitable ID for this purpose, since some BIOs will be handled by kernel
+   * threads like aio or pdflush on behalf of the process requesting the BIOs.
+   */
+ return 0; /* not implemented yet */
+}
+
+struct group_type dm_ioband_group_type[] = {
+ {"none",   NULL},
+ {"pgrp",   ioband_process_group},
+ {"pid",    ioband_process_id},
+ {"node",   ioband_node},
+ {"cpuset", ioband_cpuset},
+ {"cgroup", ioband_cgroup},
+ {"user",   ioband_uid},
+ {"uid",    ioband_uid},
+ {"gid",    ioband_gid},
+ {NULL,     NULL}
+};
diff -uprN linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband.h
linux-2.6.26-rc5-mm3/drivers/md/dm-ioband.h
--- linux-2.6.26-rc5-mm3.orig/drivers/md/dm-ioband.h 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.26-rc5-mm3/drivers/md/dm-ioband.h 2008-07-11 19:24:05.000000000 +0900
@@ -0,0 +1,190 @@
+/*
+ * Copyright (C) 2008 VA Linux Systems Japan K.K.
+ *
+ *  I/O bandwidth control
+ *
+ * This file is released under the GPL.
```

```
+ */
+
+#include <linux/version.h>
+#include <linux/wait.h>
+
+#define DEFAULT_IO_THROTTLE 4
+#define DEFAULT_IO_LIMIT 128
+#define IOBAND_NAME_MAX 31
+#define IOBAND_ID_ANY (-1)
+
+struct ioband_group;
+
+struct ioband_device {
+ struct list_head g_groups;
+ struct delayed_work     g_conductor;
+ struct workqueue_struct *g_ioband_wq;
+ struct bio_list g_urgent_bios;
+ int g_io_throttle;
+ int g_io_limit[2];
+ int g_issued[2];
+ int g_blocked;
+ spinlock_t g_lock;
+ struct mutex g_lock_device;
+ wait_queue_head_t g_waitq;
+ wait_queue_head_t g_waitq_suspend;
+ wait_queue_head_t g_waitq_flush;
+
+ int g_ref;
+ struct list_head g_list;
+ int g_flags;
+ char g_name[IOBAND_NAME_MAX + 1];
+ struct policy_type *g_policy;
+
+ /* policy dependent */
+ int (*g_can_submit)(struct ioband_group *);
+ int (*g_prepare_bio)(struct ioband_group *, struct bio *, int);
+ int (*g_restart_bios)(struct ioband_device *);
+ void (*g_hold_bio)(struct ioband_group *, struct bio *);
+ struct bio * (*g_pop_bio)(struct ioband_group *);
+ int (*g_group_ctr)(struct ioband_group *, char *);
+ void (*g_group_dtr)(struct ioband_group *);
+ int (*g_set_param)(struct ioband_group *, char *cmd, char *value);
+ int (*g_should_block)(struct ioband_group *);
+ void (*g_show)(struct ioband_group *, int *, char *, unsigned int);
+
+ /* members for weight balancing policy */
+ int g_epoch;
+ int g_weight_total;
```

```
+  /* the number of tokens which can be used in every epoch */
+ int g_token_bucket;
+  /* how many epochs tokens can be carried over */
+ int g_carryover;
+  /* how many tokens should be used for one page-sized I/O */
+ int g_token_unit;
+  /* the last group which used a token */
+ struct ioband_group *g_current;
+  /* give another group a chance to be scheduled when the rest
+     of tokens of the current group reaches this mark */
+ int g_yield_mark;
+  /* the latest group which used up its tokens */
+ struct ioband_group *g_expired;
+  /* the group which has the largest number of tokens in the
+     active groups */
+ struct ioband_group *g_dominant;
+  /* the number of unused tokens in this epoch */
+ int g_token_left;
+  /* left-over tokens from the previous epoch */
+ int g_token_extra;
+};
+
+struct ioband_group_stat {
+ unsigned long sectors;
+ unsigned long immediate;
+ unsigned long deferred;
+};
+
+struct ioband_group {
+ struct list_head c_list;
+ struct ioband_device *c_banddev;
+ struct dm_dev *c_dev;
+ struct dm_target *c_target;
+ struct bio_list c_blocked_bios;
+ struct bio_list c_prio_bios;
+ struct rb_root c_group_root;
+ struct  rb_node c_group_node;
+ int c_id; /* should be unsigned long or unsigned long long */
+ char c_name[IOBAND_NAME_MAX + 1]; /* rfu */
+ int c_blocked;
+ int c_prio_blocked;
+ wait_queue_head_t c_waitq;
+ int c_flags;
+ struct ioband_group_stat c_stat[2]; /* hold rd/wr status */
+ struct group_type *c_type;
+
+ /* members for weight balancing policy */
+ int c_weight;
```

```
+ int c_my_epoch;
+ int c_token;
+ int c_token_initial;
+ int c_limit;
+ int     c_consumed;
+
+ /* rfu */
+ /* struct bio_list c_ordered_tag_bios; */
+};
+
+#define IOBAND_URGENT 1
+
+#define DEV_BIO_BLOCKED  1
+#define DEV_SUSPENDED  2
+
+#define set_device_blocked(dp)  ((dp)->g_flags |= DEV_BIO_BLOCKED)
+#define clear_device_blocked(dp) ((dp)->g_flags &= ~DEV_BIO_BLOCKED)
+#define is_device_blocked(dp)  ((dp)->g_flags & DEV_BIO_BLOCKED)
+
+#define set_device_suspended(dp) ((dp)->g_flags |= DEV_SUSPENDED)
+#define clear_device_suspended(dp) ((dp)->g_flags &= ~DEV_SUSPENDED)
+#define is_device_suspended(dp)  ((dp)->g_flags & DEV_SUSPENDED)
+
+#define IOG_PRIO_BIO_WRITE 1
+#define IOG_PRIO_QUEUE  2
+#define IOG_BIO_BLOCKED  4
+#define IOG_GOING_DOWN  8
+#define IOG_SUSPENDED  16
+#define IOG_NEED_UP  32
+
+#define R_OK  0
+#define R_BLOCK  1
+#define R_YIELD  2
+
+#define set_group_blocked(gp)  ((gp)->c_flags |= IOG_BIO_BLOCKED)
+#define clear_group_blocked(gp)  ((gp)->c_flags &= ~IOG_BIO_BLOCKED)
+#define is_group_blocked(gp)  ((gp)->c_flags & IOG_BIO_BLOCKED)
+
+#define set_group_down(gp)  ((gp)->c_flags |= IOG_GOING_DOWN)
+#define clear_group_down(gp)  ((gp)->c_flags &= ~IOG_GOING_DOWN)
+#define is_group_down(gp)  ((gp)->c_flags & IOG_GOING_DOWN)
+
+#define set_group_suspended(gp)  ((gp)->c_flags |= IOG_SUSPENDED)
+#define clear_group_suspended(gp) ((gp)->c_flags &= ~IOG_SUSPENDED)
+#define is_group_suspended(gp)  ((gp)->c_flags & IOG_SUSPENDED)
+
+#define set_group_need_up(gp)  ((gp)->c_flags |= IOG_NEED_UP)
+#define clear_group_need_up(gp)  ((gp)->c_flags &= ~IOG_NEED_UP)
```

```
+#define group_need_up(gp)  ((gp)->c_flags & IOG_NEED_UP)
+
+#define set_prio_read(gp)  ((gp)->c_flags |= IOG_PRIO_QUEUE)
+#define clear_prio_read(gp)  ((gp)->c_flags &= ~IOG_PRIO_QUEUE)
+#define is_prio_read(gp) \
+ ((gp)->c_flags & (IOG_PRIO_QUEUE|IOG_PRIO_BIO_WRITE) == IOG_PRIO_QUEUE)
+
+#define set_prio_write(gp) \
+ ((gp)->c_flags |= (IOG_PRIO_QUEUE|IOG_PRIO_BIO_WRITE))
+#define clear_prio_write(gp) \
+ ((gp)->c_flags &= ~(IOG_PRIO_QUEUE|IOG_PRIO_BIO_WRITE))
+#define is_prio_write(gp) \
+ ((gp)->c_flags & (IOG_PRIO_QUEUE|IOG_PRIO_BIO_WRITE) == \
+  (IOG_PRIO_QUEUE|IOG_PRIO_BIO_WRITE))
+
+#define set_prio_queue(gp, direct) \
+ ((gp)->c_flags |= (IOG_PRIO_QUEUE|direct))
+#define clear_prio_queue(gp)  clear_prio_write(gp)
+#define is_prio_queue(gp)  ((gp)->c_flags & IOG_PRIO_QUEUE)
+#define prio_queue_direct(gp)  ((gp)->c_flags & IOG_PRIO_BIO_WRITE)
+
+
+struct policy_type {
+ const char *p_name;
+ int   (*p_policy_init)(struct ioband_device *, int, char **);
+};
+
+extern struct policy_type dm_ioband_policy_type[];
+
+struct group_type {
+ const char *t_name;
+ int   (*t_getid)(struct bio *);
+};
+
+extern struct group_type dm_ioband_group_type[];
+
+/* Just for debugging */
+extern long ioband_debug;
+#define dprintk(format, a...) \
+ if (ioband_debug > 0) ioband_debug--, printk(format, ##a)
```