Subject: [PATCH 1/2] dm-ioband v0.0.3: The I/O bandwidth controller: Source code patch
Posted by Ryo Tsuruta on Tue, 05 Feb 2008 10:18:46 GMT
View Forum Message <> Reply to Message

Here is the patch of dm-ioband.

Based on 2.6.24
Signed-off-by: Ryo Tsuruta <ryov@valinux.co.jp>
Signed-off-by: Hirokazu Takahashi <taka@valinux.co.jp>

```
diff -uprN linux-2.6.24.orig/drivers/md/Kconfig linux-2.6.24/drivers/md/Kconfig
--- linux-2.6.24.orig/drivers/md/Kconfig 2008-01-25 07:58:37.000000000 +0900
+++ linux-2.6.24/drivers/md/Kconfig 2008-02-05 19:09:41.000000000 +0900
@@ -288,4 +288,17 @@ 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 bandwith 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-2.6.24.orig/drivers/md/Makefile linux-2.6.24/drivers/md/Makefile
--- linux-2.6.24.orig/drivers/md/Makefile 2008-01-25 07:58:37.000000000 +0900
+++ linux-2.6.24/drivers/md/Makefile 2008-02-05 19:09:41.000000000 +0900
@@ -9,6 +9,7 @@ dm-snapshot-objs := dm-snap.o dm-excepti
 dm-mirror-objs := dm-log.o dm-raid1.o
 dm-rdac-objs := dm-mpath-rdac.o
 dm-hp-sw-objs := dm-mpath-hp-sw.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 \
@@ -41,6 +42,7 @@ obj-$(CONFIG_DM_MULTIPATH_RDAC) += dm-rd
 obj-$(CONFIG_DM_SNAPSHOT) += dm-snapshot.o
 obj-$(CONFIG_DM_MIRROR)  += dm-mirror.o
 obj-$(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-2.6.24.orig/drivers/md/dm-ioband-ctl.c linux-2.6.24/drivers/md/dm-ioband-ctl.c
--- linux-2.6.24.orig/drivers/md/dm-ioband-ctl.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.24/drivers/md/dm-ioband-ctl.c 2008-02-05 19:09:41.000000000 +0900
@@ -0,0 +1,947 @@
+/*
+ * 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 "dm.h"
+#include "dm-bio-list.h"
+#include "dm-ioband.h"
+
+#define DM_MSG_PREFIX "ioband"
+
+static LIST_HEAD(ioband_device_list);
+/* to protect ioband_device_list */
+static DEFINE_SPINLOCK(ioband_devicelist_lock);
+
+#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
+static void ioband_conduct(void *);
+#else
+static void ioband_conduct(struct work_struct *);
+#endif
+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);
+
+int ioband_debug; /* just for debugging */
+
+static void policy_init(struct ioband_device *dp, char *name)
+{
+ struct policy_type *p;
+ for (p = dm_ioband_policy_type; (p->p_name); p++) {
```

```
+  if (!strcmp(name, p->p_name))
+   break;
+ }
+
+ p->p_policy_init(dp);
+ if (!dp->g_hold_bio)
+  dp->g_hold_bio = ioband_hold_bio;
+ if (!dp->g_pop_bio)
+  dp->g_pop_bio = ioband_pop_bio;
+}
+
+static struct ioband_device *alloc_ioband_device(int devgroup_id, char *name,
+    int io_throttle, int io_limit)
+
+{
+ struct ioband_device *dp = NULL;
+ struct ioband_device *p;
+ struct ioband_device *new;
+ unsigned long flags;
+
+ new = kzalloc(sizeof(struct ioband_device), GFP_KERNEL);
+ if (!new)
+  goto try_to_find;
+
+ /*
+  * 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) {
+  kfree(new);
+  new = NULL;
+  goto try_to_find;
+ }
+
+#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
+ INIT_WORK(&new->g_conductor, ioband_conduct, new);
+#else
+ INIT_WORK(&new->g_conductor, ioband_conduct);
+#endif
+ INIT_LIST_HEAD(&new->g_groups);
+ INIT_LIST_HEAD(&new->g_list);
+ spin_lock_init(&new->g_lock);
+ new->g_devgroup = devgroup_id;
+ new->g_io_throttle = io_throttle;
+ new->g_io_limit = io_limit;
+ new->g_plug_bio = 0;
+ new->g_issued = 0;
```

```
+ new->g_blocked = 0;
+ new->g_ref = 0;
+ new->g_flags = 0;
+ memset(new->g_name, 0, sizeof(new->g_name));
+ new->g_hold_bio = NULL;
+ new->g_pop_bio = NULL;
+ init_waitqueue_head(&new->g_waitq);
+
+try_to_find:
+ spin_lock_irqsave(&ioband_devicelist_lock, flags);
+ list_for_each_entry(p, &ioband_device_list, g_list) {
+  if (p->g_devgroup == devgroup_id) {
+   dp = p;
+   break;
+  }
+ }
+ if (!dp && (new)) {
+  policy_init(new, name);
+  dp = new;
+  new = NULL;
+  list_add_tail(&dp->g_list, &ioband_device_list);
+ }
+ spin_unlock_irqrestore(&ioband_devicelist_lock, flags);
+
+ if (new) {
+  destroy_workqueue(new->g_ioband_wq);
+  kfree(new);
+ }
+
+ return dp;
+}
+
+static inline void release_ioband_device(struct ioband_device *dp)
+{
+ unsigned long flags;
+
+ spin_lock_irqsave(&ioband_devicelist_lock, flags);
+ if (!list_empty(&dp->g_groups)) {
+  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 struct ioband_group *ioband_group_find(struct ioband_group *head,int id)
```

```
+{
+ struct ioband_group *p;
+ struct ioband_group *gp = NULL;
+
+ list_for_each_entry(p, &head->c_group_list, c_group_list) {
+  if (p->c_id == id || id == IOBAND_ID_ANY)
+   gp = p;
+ }
+ return gp;
+}
+
+static int ioband_group_init(struct ioband_group *gp,
+  struct ioband_group *head, struct ioband_device *dp, int id)
+{
+ unsigned long flags;
+
+ INIT_LIST_HEAD(&gp->c_list);
+ bio_list_init(&gp->c_blocked_bios);
+ gp->c_id = id; /* should be verified */
+ gp->c_blocked = 0;
+ memset(gp->c_stat, 0, sizeof(gp->c_stat));
+ init_waitqueue_head(&gp->c_waitq);
+ gp->c_flags = 0;
+
+ INIT_LIST_HEAD(&gp->c_group_list);
+
+ 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;
+ }
+ dp->g_ref++;
+ list_add_tail(&gp->c_list, &dp->g_groups);
+
+ dp->g_group_ctr(gp);
+
+ if (head) {
+  list_add_tail(&gp->c_group_list, &head->c_group_list);
+  gp->c_dev = head->c_dev;
+  gp->c_target = head->c_target;
+ }
+
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ return 0;
```

```
+}
+
+static inline void ioband_group_release(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ list_del(&gp->c_list);
+ list_del(&gp->c_group_list);
+ dp->g_ref--;
+ 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(group);
+ ioband_group_release(gp);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+}
+
+static void ioband_group_stop(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ unsigned long flags;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ set_group_down(gp);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ queue_work(dp->g_ioband_wq, &dp->g_conductor);
+ flush_workqueue(dp->g_ioband_wq);
+}
+
+static void ioband_group_stop_all(struct ioband_group *head, int suspend)
+{
+ struct ioband_device *dp = head->c_banddev;
+ struct ioband_group *p;
+ unsigned long flags;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ list_for_each_entry(p, &head->c_group_list, c_group_list) {
+  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_work(dp->g_ioband_wq, &dp->g_conductor);
+ 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;
+ unsigned long flags;
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ list_for_each_entry(p, &head->c_group_list, c_group_list) {
+  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);
+}
+
+/*
+ * Create a new band device:
+ *   parameters:  <device> <device-group-id> [<io_throttle>] [<io_limit>]
+ */
+static int ioband_ctr(struct dm_target *ti, unsigned int argc, char **argv)
+{
+ struct ioband_group *gp;
+ struct ioband_device *dp;
+ int io_throttle = DEFAULT_IO_THROTTLE;
+ int io_limit = DEFAULT_IO_LIMIT;
+ int devgroup_id;
+ int val;
+ int r = 0;
+
+ if (argc < 2) {
```

```
+  ti->error = "Requires 2 or more arguments";
+  return -EINVAL;
+ }
+
+ gp = kzalloc(sizeof(struct ioband_group), GFP_KERNEL);
+ if (!gp) {
+  ti->error = "Cannot allocate memory for bandgroup";
+  return -ENOMEM;
+ }
+
+ val = simple_strtol(argv[1], NULL, 0);
+ if (val < 0) {
+  ti->error = "Device Group ID # is too large";
+  r = -EINVAL;
+  goto error;
+ }
+ devgroup_id = val;
+ dprintk(KERN_ERR "ioband_ctr device group id:%d\n", val);
+
+ if (argc >= 3) {
+  val = simple_strtol(argv[2], NULL, 0);
+  if (val > 0)
+   io_throttle = val;
+  dprintk(KERN_ERR "ioband_ctr ioqueue_low:%d\n", io_throttle);
+ }
+ if (argc >= 4) {
+  val = simple_strtol(argv[3], NULL, 0);
+  if (val > 0)
+   io_limit = val;
+  dprintk(KERN_ERR "ioband_ctr ioqueue_high:%d\n", io_limit);
+ }
+ if (io_limit < io_throttle)
+  io_limit = io_throttle;
+
+ if (dm_get_device(ti, argv[0], 0, ti->len,
+     dm_table_get_mode(ti->table), &gp->c_dev)) {
+  ti->error = "band: device lookup failed";
+  r = -EINVAL;
+  goto error;
+ }
+
+ dp = alloc_ioband_device(devgroup_id, "default",io_throttle, io_limit);
+ if (!dp) {
+  ti->error = "Cannot allocate memory for banddevice";
+  r = -ENOMEM;
+  goto error2;
+ }
+
```

```
+ ioband_group_init(gp, NULL, dp, IOBAND_ID_ANY);
+ gp->c_getid = dm_ioband_group_type[0].t_getid;
+
+ ti->private = gp;
+
+ return 0;
+
+error2:
+ dm_put_device(ti, gp->c_dev);
+error:
+ kfree(gp);
+ return r;
+}
+
+static void ioband_dtr(struct dm_target *ti)
+{
+ struct ioband_group *gp = ti->private;
+ struct ioband_device *dp = gp->c_banddev;
+
+ ioband_group_stop_all(gp, 0);
+ dm_put_device(ti, gp->c_dev);
+ ioband_group_destroy_all(gp);
+ 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 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) {
+  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) {
+  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 inline void do_nothing(void) {}
+
+static inline void prevent_burst_bios(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (!current->mm) {
+  /*
+   * 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 void prepare_to_issue(struct ioband_group *gp, struct bio *bio)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ dp->g_prepare_bio(gp, bio);
+ dp->g_issued++;
+ if (dp->g_issued >= dp->g_io_limit)
+  dp->g_plug_bio = 1;
+}
+
+static inline int room_for_bio(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ return !dp->g_plug_bio || is_group_down(gp);
+}
+
+static inline void hold_bio(struct ioband_group *gp, struct bio *bio)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ dp->g_blocked++;
+ gp->c_blocked++;
+ gp->c_stat[bio_data_dir(bio)].deferred++;
+ dp->g_hold_bio(gp, bio);
+}
+
+static inline int release_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;
+
+ while (dp->g_can_submit(gp) && gp->c_blocked) {
+  if (!room_for_bio(gp))
+   return 1;
+  bio = dp->g_pop_bio(gp);
+  if (!bio)
+   return 0;
```

```
+  dp->g_blocked--;
+  gp->c_blocked--;
+  if (!gp->c_blocked)
+   resume_to_accept_bios(gp);
+  prepare_to_issue(gp, bio);
+  if (is_group_suspended(gp))
+   bio_list_add(pushback_list, bio);
+  else
+   bio_list_add(issue_list, bio);
+ }
+
+ return 0;
+}
+
+static inline struct ioband_group *ioband_group_get(
+   struct ioband_group *head, struct bio *bio)
+{
+ struct ioband_group *gp;
+
+ if (!head->c_getid)
+  return head;
+
+ gp = ioband_group_find(head, head->c_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_group_stat *bws;
+ struct ioband_device *dp = gp->c_banddev;
+ unsigned long flags;
+
+#if 0 /* not supported yet */
+ if (bio_barrier(bio))
+  return -EOPNOTSUPP;
+#endif
+
+ spin_lock_irqsave(&dp->g_lock, flags);
+ gp = ioband_group_get(gp, bio);
```

```
+ prevent_burst_bios(gp);
+ 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;
+retry:
+ if (!gp->c_blocked && room_for_bio(gp)) {
+  if (dp->g_can_submit(gp)) {
+   bws = &gp->c_stat[bio_data_dir(bio)];
+   bws->sectors += bio_sectors(bio);
+   bws->immediate++;
+   prepare_to_issue(gp, bio);
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   return DM_MAPIO_REMAPPED;
+  } else if (dp->g_issued < dp->g_io_throttle) {
+   dprintk(KERN_ERR "ioband_map: token expired "
+     "gp:%p bio:%p\n", gp, bio);
+   if (dp->g_restart_bios(dp))
+    goto retry;
+  }
+ }
+ hold_bio(gp, bio);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ return DM_MAPIO_SUBMITTED;
+}
+
+/*
+ * Select the best group to resubmit its BIOs.
+ */
+static inline 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(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.
+ */
+#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
+static void ioband_conduct(void *p)
+{
+ struct ioband_device *dp = (struct ioband_device *)p;
+#else
+static void ioband_conduct(struct work_struct *work)
+{
+ struct ioband_device *dp =
+  container_of(work, struct ioband_device, g_conductor);
+#endif
+ 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);
+retry:
+ while (dp->g_blocked && (gp = choose_best_group(dp)))
+  release_bios(gp, &issue_list, &pushback_list);
+
+ if (dp->g_blocked && dp->g_issued < dp->g_io_throttle) {
+  dprintk(KERN_ERR "ioband_conduct: token expired gp:%p\n", gp);
+  if (dp->g_restart_bios(dp))
+   goto retry;
+ }
+
+ 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)))
+#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
+  bio_endio(bio, bio->bi_size, -EIO);
```

```
+#else
+  bio_endio(bio, -EIO);
+#endif
+}
+
+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);
+ if (dp->g_issued <= dp->g_io_throttle)
+  dp->g_plug_bio = 0;
+ dp->g_issued--;
+
+ /*
+  * Todo: It would be better to introduce high/low water marks here
+  *   not to kick the workqueues so often.
+  */
+ if (dp->g_blocked && !dp->g_plug_bio)
+  queue_work(dp->g_ioband_wq, &dp->g_conductor);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ return r;
+}
+
+static void ioband_presuspend(struct dm_target *ti)
+{
+ struct ioband_group *gp = ti->private;
+ ioband_group_stop_all(gp, 1);
+}
+
+static void ioband_resume(struct dm_target *ti)
+{
+ struct ioband_group *gp = ti->private;
+ ioband_group_resume_all(gp);
+}
```

```
+
+
+static void ioband_group_status(struct ioband_group *gp, int *szp, char *result,
+    unsigned int maxlen)
+{
+ struct ioband_group_stat *bws;
+ unsigned long reqs;
+ int i, sz = *szp; /* used in DMEMIT() */
+
+ DMEMIT(" %d", gp->c_id);
+ for (i = 0; i < 2; i++) {
+  bws = &gp->c_stat[i];
+  reqs = bws->immediate + bws->deferred;
+  DMEMIT(" %lu %lu %lu",
+   bws->immediate + bws->deferred, bws->deferred,
+   bws->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;
+ int sz = 0; /* used in DMEMIT() */
+ unsigned long flags;
+
+ switch (type) {
+ case STATUSTYPE_INFO:
+  spin_lock_irqsave(&dp->g_lock, flags);
+  DMEMIT("%d", dp->g_devgroup);
+  ioband_group_status(gp, &sz, result, maxlen);
+  list_for_each_entry(p, &gp->c_group_list, c_group_list)
+   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 %d %d %d", gp->c_dev->name, dp->g_devgroup,
+      dp->g_io_throttle, dp->g_io_limit);
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  break;
+ }
+
+ 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 (!list_empty(&gp->c_group_list)) {
+  spin_unlock_irqrestore(&dp->g_lock, flags);
+  return -EBUSY;
+ }
+ gp->c_getid = t->t_getid;
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+
+ return 0;
+}
+
+static inline int split_string(char *s, char **v)
+{
+ int id = IOBAND_ID_ANY;
+ char *p, *q;
+
+ p = strsep(&s, "=:,");
+ q = strsep(&s, "=:,");
+ if (!q) {
+  *v = p;
+ } else {
+  id = simple_strtol(p, NULL, 0);
+  *v = q;
+ }
+ return id;
+}
+
+static int ioband_set_param(struct ioband_group *gp, char *cmd, char *value)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ char *val_str;
+ int id;
+ unsigned long flags;
```

```
+ int r;
+
+ id = split_string(value, &val_str);
+
+ 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=%d 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)
+{
+ 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;
+ }
+ sub_gp = kzalloc(sizeof(struct ioband_group), GFP_KERNEL);
+ if (!sub_gp)
+  return -ENOMEM;
+
+ r = ioband_group_init(sub_gp, gp, dp, id);
+ 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);
+ spin_unlock_irqrestore(&dp->g_lock, flags);
+ if (!sub_gp) {
+  DMWARN("ioband_group_detach: invalid id:%d", id);
+  return -EINVAL;
+ }
+ ioband_group_stop(sub_gp);
+ spin_lock_irqsave(&dp->g_lock, flags);
+ ioband_group_release(sub_gp);
+ 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;
+ int val = 0;
+ 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));
+  list_for_each_entry(p, &gp->c_group_list, c_group_list)
+   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")) {
+  ioband_debug = simple_strtol(argv[1], NULL, 0);
+  if (ioband_debug < 0) ioband_debug = 0;
+  return 0;
+ } else if (!strcmp(argv[0], "io_throttle")) {
+  val = simple_strtol(argv[1], NULL, 0);
+  spin_lock_irqsave(&dp->g_lock, flags);
+  if (val <= 0 || val >= dp->g_io_limit) {
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   return -EINVAL;
+  }
+  dp->g_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")) {
+  val = simple_strtol(argv[1], NULL, 0);
+  spin_lock_irqsave(&dp->g_lock, flags);
+  if (val <= dp->g_io_throttle) {
+   spin_unlock_irqrestore(&dp->g_lock, flags);
+   return -EINVAL;
+  }
+  dp->g_io_limit = 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")) {
+  int id = simple_strtol(argv[1], NULL, 0);
+  return ioband_group_attach(gp, id);
+ } else if (!strcmp(argv[0], "detach")) {
+  int id = simple_strtol(argv[1], NULL, 0);
+  return ioband_group_detach(gp, id);
+ } 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 struct target_type ioband_target = {
+ .name      = "ioband",
+ .module      = THIS_MODULE,
+ .version     = {0, 0, 3},
+ .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-2.6.24.orig/drivers/md/dm-ioband-policy.c
linux-2.6.24/drivers/md/dm-ioband-policy.c
--- linux-2.6.24.orig/drivers/md/dm-ioband-policy.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.24/drivers/md/dm-ioband-policy.c 2008-02-05 19:09:41.000000000 +0900
@@ -0,0 +1,202 @@
+/*
```

```
+ * 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 "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.
+ */
+#define DEFAULT_WEIGHT 100
+#define DEFAULT_TOKENBASE 2048
+#define IOBAND_IOPRIO_BASE 100
+
+static int proceed_global_epoch(struct ioband_device *dp)
+{
+ dp->g_epoch++;
+#if 0 /* this will also work correctly */
+ if (dp->g_blocked)
+  queue_work(dp->g_ioband_wq, &dp->g_conductor);
+ return 0;
+#endif
+ dprintk(KERN_ERR "proceed_epoch %d --> %d\n",
+      dp->g_epoch-1, dp->g_epoch);
+ return 1;
+}
+
+static inline int proceed_epoch(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ if (gp->c_my_epoch != dp->g_epoch) {
+  gp->c_my_epoch = dp->g_epoch;
+  return 1;
+ }
+ return 0;
```

```
+}
+
+static inline int iopriority(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ int iopri;
+
+ iopri = gp->c_token*IOBAND_IOPRIO_BASE/gp->c_token_init_value + 1;
+ if (gp->c_my_epoch != dp->g_epoch)
+  iopri += IOBAND_IOPRIO_BASE;
+ if (is_group_down(gp))
+  iopri += IOBAND_IOPRIO_BASE*2;
+
+ return iopri;
+}
+
+static int is_token_left(struct ioband_group *gp)
+{
+ if (gp->c_token > 0)
+  return iopriority(gp);
+
+ if (proceed_epoch(gp) || is_group_down(gp)) {
+  gp->c_token = gp->c_token_init_value;
+  dprintk(KERN_ERR "refill token: gp:%p token:%d\n",
+          gp, gp->c_token);
+  return iopriority(gp);
+ }
+ return 0;
+}
+
+static void prepare_token(struct ioband_group *gp, struct bio *bio)
+{
+ gp->c_token--;
+}
+
+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;
+
+ list_for_each_entry(p, &dp->g_groups, c_list) {
+  /* Fixme: it might overflow */
+  p->c_token = p->c_token_init_value =
+      dp->g_token_base*p->c_weight/dp->g_weight_total + 1;
+  p->c_limit =
```

```
+     dp->g_io_limit*p->c_weight/dp->g_weight_total + 1;
+ }
+}
+
+static int policy_weight_ctr(struct ioband_group *gp)
+{
+ struct ioband_device *dp = gp->c_banddev;
+
+ gp->c_my_epoch = dp->g_epoch;
+ gp->c_weight = 0;
+ set_weight(gp, DEFAULT_WEIGHT);
+ return 0;
+}
+
+static void policy_weight_dtr(struct ioband_group *gp)
+{
+ set_weight(gp, 0);
+}
+
+static int policy_weight_param(struct ioband_group *gp, char *cmd, char *value)
+{
+ struct ioband_device *dp = gp->c_banddev;
+ int val = simple_strtol(value, NULL, 0);
+ int r = 0;
+
+ if (!strcmp(cmd, "weight")) {
+  if (val > 0)
+   set_weight(gp, val);
+  else
+   r = -EINVAL;
+ } else if (!strcmp(cmd, "token")) {
+  if (val > 0) {
+   dp->g_token_base = val;
+   set_weight(gp, gp->c_weight);
+  } else
+   r = -EINVAL;
+ } else if (!strcmp(cmd, "io_limit")) {
+  set_weight(gp, gp->c_weight);
+ } else {
+  r = -EINVAL;
+ }
+ return r;
+}
+
+static int is_queue_full(struct ioband_group *gp)
+{
+ return gp->c_blocked >= gp->c_limit;
+}
```

```
+
+
+/*
+ *  <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.
+ */
+static void policy_weight_init(struct ioband_device *dp)
+{
+ dp->g_can_submit = is_token_left;
+ dp->g_prepare_bio = prepare_token;
+ dp->g_restart_bios = proceed_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_token_base = DEFAULT_TOKENBASE;
+ dp->g_epoch = 0;
+ dp->g_weight_total = 0;
+}
+/* weight balancing policy. --- End --- */
+
+
+static void policy_default_init(struct ioband_device *dp) /*XXX*/
```

```
+{
+ policy_weight_init(dp); /* temp */
+}
+
+struct policy_type dm_ioband_policy_type[] = {
+ {"default", policy_default_init},
+ {"weight", policy_weight_init},
+ {NULL,     policy_default_init}
+};
diff -uprN linux-2.6.24.orig/drivers/md/dm-ioband-type.c linux-2.6.24/drivers/md/dm-ioband-type.c
--- linux-2.6.24.orig/drivers/md/dm-ioband-type.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.24/drivers/md/dm-ioband-type.c 2008-02-05 19:09:41.000000000 +0900
@@ -0,0 +1,80 @@
+/*
+ * 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)
+{
+#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
+ return (int)process_group(current);
+#else
+ return (int)task_pgrp_nr(current);
+#endif
```

```
+}
+
+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-2.6.24.orig/drivers/md/dm-ioband.h linux-2.6.24/drivers/md/dm-ioband.h
--- linux-2.6.24.orig/drivers/md/dm-ioband.h 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.24/drivers/md/dm-ioband.h 2008-02-05 19:09:41.000000000 +0900
```

```
@@ -0,0 +1,129 @@
+/*
+ * 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 work_struct g_conductor;
+ struct workqueue_struct *g_ioband_wq;
+ int g_io_throttle;
+ int g_io_limit;
+ int g_plug_bio;
+ int g_issued;
+ int g_blocked;
+ spinlock_t g_lock;
+ wait_queue_head_t g_waitq;
+
+ int g_devgroup;
+ int g_ref;  /* just for debugging */
+ struct list_head g_list;
+ int g_flags;
+ char g_name[IOBAND_NAME_MAX + 1]; /* rfu */
+
+ /* policy dependent */
+ int (*g_can_submit)(struct ioband_group *);
+ void (*g_prepare_bio)(struct ioband_group *, struct bio *);
+ 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 *);
+ 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 *);
+
```

```
+ /* members for weight balancing policy */
+ int g_epoch;
+ int g_weight_total;
+ int g_token_base;
+
+};
+
+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 list_head c_group_list;
+ int c_id; /* should be unsigned long or unsigned long long */
+ char c_name[IOBAND_NAME_MAX + 1]; /* rfu */
+ int c_blocked;
+ wait_queue_head_t c_waitq;
+ int c_flags;
+ struct ioband_group_stat c_stat[2]; /* hold rd/wr status */
+
+ /* type dependent */
+ int (*c_getid)(struct bio *);
+
+ /* members for weight balancing policy */
+ int c_weight;
+ int c_my_epoch;
+ int c_token;
+ int c_token_init_value;
+ int c_limit;
+
+ /* rfu */
+ /* struct bio_list c_ordered_tag_bios; */
+};
+
+#define DEV_BIO_BLOCKED 1
+
+#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 IOG_BIO_BLOCKED 1
+#define IOG_GOING_DOWN 2
+#define IOG_SUSPENDED 4
+
+#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)
+
+struct policy_type {
+ const char *p_name;
+ void   (*p_policy_init)(struct ioband_device *);
+};
+
+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 int ioband_debug;
+#define dprintk(format, a...) \
+ if (ioband_debug > 0) ioband_debug--, printk(format, ##a)
```