Subject: [PATCH 3/3] i/o accounting and control
Posted by Andrea Righi on Fri, 20 Jun 2008 10:05:35 GMT
View Forum Message <> Reply to Message

Apply the io-throttle controller to the opportune kernel functions. Both
accounting and throttling functionalities are performed by
cgroup_io_throttle().

Signed-off-by: Andrea Righi <righi.andrea@gmail.com>
---
 block/blk-core.c | 2 ++
 fs/buffer.c | 10 ++++++++++
 fs/direct-io.c | 3 +++
 mm/page-writeback.c | 9 +++++++++
 mm/readahead.c | 5 +++++
 5 files changed, 29 insertions(+), 0 deletions(-)

diff --git a/block/blk-core.c b/block/blk-core.c
index 1905aab..8eddef5 100644
--- a/block/blk-core.c
+++ b/block/blk-core.c
@@ -26,6 +26,7 @@
 #include <linux/swap.h>
 #include <linux/writeback.h>
 #include <linux/task_io_accounting_ops.h>
+#include <linux/blk-io-throttle.h>
 #include <linux/interrupt.h>
 #include <linux/cpu.h>
 #include <linux/blktrace_api.h>
@@ -1482,6 +1483,7 @@ void submit_bio(int rw, struct bio *bio)
    count_vm_events(PGPGOUT, count);
   } else {
    task_io_account_read(bio->bi_size);
+   cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
    count_vm_events(PGPGIN, count);
   }

diff --git a/fs/buffer.c b/fs/buffer.c
index a073f3f..c63dfe7 100644
--- a/fs/buffer.c
+++ b/fs/buffer.c
@@ -35,6 +35,7 @@
 #include <linux/suspend.h>
 #include <linux/buffer_head.h>
 #include <linux/task_io_accounting_ops.h>
+#include <linux/blk-io-throttle.h>
 #include <linux/bio.h>
 #include <linux/notifier.h>

```
 #include <linux/cpu.h>
@@ -700,6 +701,9 @@ EXPORT_SYMBOL(mark_buffer_dirty_inode);
 static int __set_page_dirty(struct page *page,
   struct address_space *mapping, int warn)
 {
+ struct block_device *bdev = NULL;
+ size_t cgroup_io_acct = 0;
+
  if (unlikely(!mapping))
   return !TestSetPageDirty(page);

@@ -711,16 +715,22 @@ static int __set_page_dirty(struct page *page,
   WARN_ON_ONCE(warn && !PageUptodate(page));

   if (mapping_cap_account_dirty(mapping)) {
+   bdev = (mapping->host &&
+    mapping->host->i_sb->s_bdev) ?
+    mapping->host->i_sb->s_bdev : NULL;
+
    __inc_zone_page_state(page, NR_FILE_DIRTY);
    __inc_bdi_stat(mapping->backing_dev_info,
     BDI_RECLAIMABLE);
    task_io_account_write(PAGE_CACHE_SIZE);
+   cgroup_io_acct = PAGE_CACHE_SIZE;
   }
   radix_tree_tag_set(&mapping->page_tree,
    page_index(page), PAGECACHE_TAG_DIRTY);
  }
  write_unlock_irq(&mapping->tree_lock);
  __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
+ cgroup_io_throttle(bdev, cgroup_io_acct);

  return 1;
 }
diff --git a/fs/direct-io.c b/fs/direct-io.c
index 9e81add..fe991ac 100644
--- a/fs/direct-io.c
+++ b/fs/direct-io.c
@@ -35,6 +35,7 @@
 #include <linux/buffer_head.h>
 #include <linux/rwsem.h>
 #include <linux/uio.h>
+#include <linux/blk-io-throttle.h>
 #include <asm/atomic.h>

 /*
@@ -666,6 +667,8 @@ submit_page_section(struct dio *dio, struct page *page,
   /*
```

```
    * Read accounting is performed in submit_bio()
    */
+   struct block_device *bdev = dio->bio ? dio->bio->bi_bdev : NULL;
+   cgroup_io_throttle(bdev, len);
    task_io_account_write(len);
   }

diff --git a/mm/page-writeback.c b/mm/page-writeback.c
index 789b6ad..a2b820d 100644
--- a/mm/page-writeback.c
+++ b/mm/page-writeback.c
@@ -23,6 +23,7 @@
 #include <linux/init.h>
 #include <linux/backing-dev.h>
 #include <linux/task_io_accounting_ops.h>
+#include <linux/blk-io-throttle.h>
 #include <linux/blkdev.h>
 #include <linux/mpage.h>
 #include <linux/rmap.h>
@@ -1077,6 +1078,8 @@ int __set_page_dirty_nobuffers(struct page *page)
   if (!TestSetPageDirty(page)) {
    struct address_space *mapping = page_mapping(page);
    struct address_space *mapping2;
+   struct block_device *bdev = NULL;
+   size_t cgroup_io_acct = 0;

    if (!mapping)
     return 1;
@@ -1087,10 +1090,15 @@ int __set_page_dirty_nobuffers(struct page *page)
     BUG_ON(mapping2 != mapping);
     WARN_ON_ONCE(!PagePrivate(page) && !PageUptodate(page));
     if (mapping_cap_account_dirty(mapping)) {
+     bdev = (mapping->host &&
+      mapping->host->i_sb->s_bdev) ?
+      mapping->host->i_sb->s_bdev : NULL;
+
      __inc_zone_page_state(page, NR_FILE_DIRTY);
      __inc_bdi_stat(mapping->backing_dev_info,
       BDI_RECLAIMABLE);
      task_io_account_write(PAGE_CACHE_SIZE);
+     cgroup_io_acct = PAGE_CACHE_SIZE;
     }
     radix_tree_tag_set(&mapping->page_tree,
      page_index(page), PAGECACHE_TAG_DIRTY);
@@ -1100,6 +1108,7 @@ int __set_page_dirty_nobuffers(struct page *page)
     /* !PageAnon && !swapper_space */
     __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
    }
```

```
+  cgroup_io_throttle(bdev, cgroup_io_acct);
   return 1;
  }
  return 0;
diff --git a/mm/readahead.c b/mm/readahead.c
index d8723a5..dff6b02 100644
--- a/mm/readahead.c
+++ b/mm/readahead.c
@@ -14,6 +14,7 @@
 #include <linux/blkdev.h>
 #include <linux/backing-dev.h>
 #include <linux/task_io_accounting_ops.h>
+#include <linux/blk-io-throttle.h>
 #include <linux/pagevec.h>
 #include <linux/pagemap.h>

@@ -58,6 +59,9 @@ int read_cache_pages(struct address_space *mapping, struct list_head
*pages,
    int (*filler)(void *, struct page *), void *data)
 {
  struct page *page;
+ struct block_device *bdev =
+  (mapping->host && mapping->host->i_sb->s_bdev) ?
+  mapping->host->i_sb->s_bdev : NULL;
  int ret = 0;

  while (!list_empty(pages)) {
@@ -76,6 +80,7 @@ int read_cache_pages(struct address_space *mapping, struct list_head
*pages,
    break;
   }
   task_io_account_read(PAGE_CACHE_SIZE);
+  cgroup_io_throttle(bdev, PAGE_CACHE_SIZE);
  }
  return ret;
 }
--
1.5.4.3
```

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 3/3] i/o accounting and control
Posted by akpm on Thu, 26 Jun 2008 00:41:08 GMT

On Fri, 20 Jun 2008 12:05:35 +0200
Andrea Righi <righi.andrea@gmail.com> wrote:

> Apply the io-throttle controller to the opportune kernel functions. Both
> accounting and throttling functionalities are performed by
> cgroup_io_throttle().
>
> Signed-off-by: Andrea Righi <righi.andrea@gmail.com>
> ---
> block/blk-core.c | 2 ++
> fs/buffer.c | 10 ++++++++++
> fs/direct-io.c | 3 +++
> mm/page-writeback.c | 9 +++++++++
> mm/readahead.c | 5 +++++
> 5 files changed, 29 insertions(+), 0 deletions(-)
>
> diff --git a/block/blk-core.c b/block/blk-core.c
> index 1905aab..8eddef5 100644
> --- a/block/blk-core.c
> +++ b/block/blk-core.c
> @@ -26,6 +26,7 @@
> #include <linux/swap.h>
> #include <linux/writeback.h>
> #include <linux/task_io_accounting_ops.h>
> +#include <linux/blk-io-throttle.h>
> #include <linux/interrupt.h>
> #include <linux/cpu.h>
> #include <linux/blktrace_api.h>
> @@ -1482,6 +1483,7 @@ void submit_bio(int rw, struct bio *bio)
>   count_vm_events(PGPGOUT, count);
>   } else {
>   task_io_account_read(bio->bi_size);
> +   cgroup_io_throttle(bio->bi_bdev, bio->bi_size);

We do this on every submit_bio(READ)? Ow.

I guess it's not _hugely_ expensive, but the lengthy pointer hop in
task_subsys_state() is going to cost.

All this could be made cheaper if we were to reduce the sampling rate:
only call cgroup_io_throttle() on each megabyte of IO (for example).

```
 current->amount_of_io += bio->bi_size;
 if (current->amount_of_io > 1024*1024) {
  cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
  current->amount_of_io -= 1024 * 1024;
 }
```

but perhaps that has the potential to fail to throttle correctly when
accessing multiple devices, dunno.

Bear in mind that tasks such as pdflush and kswapd will do reads when
performing writeout (indirect blocks).

```
>     count_vm_events(PGPGIN, count);
>    }
>
> diff --git a/fs/buffer.c b/fs/buffer.c
> index a073f3f..c63dfe7 100644
> --- a/fs/buffer.c
> +++ b/fs/buffer.c
> @@ -35,6 +35,7 @@
>  #include <linux/suspend.h>
>  #include <linux/buffer_head.h>
>  #include <linux/task_io_accounting_ops.h>
> +#include <linux/blk-io-throttle.h>
>  #include <linux/bio.h>
>  #include <linux/notifier.h>
>  #include <linux/cpu.h>
> @@ -700,6 +701,9 @@ EXPORT_SYMBOL(mark_buffer_dirty_inode);
>  static int __set_page_dirty(struct page *page,
>     struct address_space *mapping, int warn)
>  {
> + struct block_device *bdev = NULL;
> + size_t cgroup_io_acct = 0;
> +
>   if (unlikely(!mapping))
>     return !TestSetPageDirty(page);
>
> @@ -711,16 +715,22 @@ static int __set_page_dirty(struct page *page,
>     WARN_ON_ONCE(warn && !PageUptodate(page));
>
>     if (mapping_cap_account_dirty(mapping)) {
> +    bdev = (mapping->host &&
> +     mapping->host->i_sb->s_bdev) ?
> +     mapping->host->i_sb->s_bdev : NULL;
> +
>      __inc_zone_page_state(page, NR_FILE_DIRTY);
>      __inc_bdi_stat(mapping->backing_dev_info,
>        BDI_RECLAIMABLE);
>      task_io_account_write(PAGE_CACHE_SIZE);
> +    cgroup_io_acct = PAGE_CACHE_SIZE;
>     }
>     radix_tree_tag_set(&mapping->page_tree,
```

>   page_index(page), PAGECACHE_TAG_DIRTY);
>   }
>   write_unlock_irq(&mapping->tree_lock);
>   __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
> + cgroup_io_throttle(bdev, cgroup_io_acct);

Ah, and here we see the reason for the __cant_sleep() hack.

It doesn't work, sorry. in_atomic() will return false when called
under spinlock when CONFIG_PREEMPT=n. Your code will call
schedule_timeout_killable() under spinlock and is deadlockable.

We need to think of something smarter here. This problem has already
been largely solved at the balance_dirty_pages() level. Perhaps that
is where the io-throttling should be cutting in?

>   return 1;
> }
> diff --git a/fs/direct-io.c b/fs/direct-io.c
> index 9e81add..fe991ac 100644
> --- a/fs/direct-io.c
> +++ b/fs/direct-io.c
> @@ -35,6 +35,7 @@
>   #include <linux/buffer_head.h>
>   #include <linux/rwsem.h>
>   #include <linux/uio.h>
> +#include <linux/blk-io-throttle.h>
>   #include <asm/atomic.h>
>
>   /*
> @@ -666,6 +667,8 @@ submit_page_section(struct dio *dio, struct page *page,
>   /*
>   * Read accounting is performed in submit_bio()
>   */
> + struct block_device *bdev = dio->bio ? dio->bio->bi_bdev : NULL;
> + cgroup_io_throttle(bdev, len);
>   task_io_account_write(len);
>   }
>
> diff --git a/mm/page-writeback.c b/mm/page-writeback.c
> index 789b6ad..a2b820d 100644
> --- a/mm/page-writeback.c
> +++ b/mm/page-writeback.c
> @@ -23,6 +23,7 @@
>   #include <linux/init.h>
>   #include <linux/backing-dev.h>
>   #include <linux/task_io_accounting_ops.h>
> +#include <linux/blk-io-throttle.h>

```
>  #include <linux/blkdev.h>
>  #include <linux/mpage.h>
>  #include <linux/rmap.h>
> @@ -1077,6 +1078,8 @@ int __set_page_dirty_nobuffers(struct page *page)
>   if (!TestSetPageDirty(page)) {
>    struct address_space *mapping = page_mapping(page);
>    struct address_space *mapping2;
> +  struct block_device *bdev = NULL;
> +  size_t cgroup_io_acct = 0;
>
>    if (!mapping)
>     return 1;
> @@ -1087,10 +1090,15 @@ int __set_page_dirty_nobuffers(struct page *page)
>     BUG_ON(mapping2 != mapping);
>     WARN_ON_ONCE(!PagePrivate(page) && !PageUptodate(page));
>     if (mapping_cap_account_dirty(mapping)) {
> +    bdev = (mapping->host &&
> +     mapping->host->i_sb->s_bdev) ?
> +     mapping->host->i_sb->s_bdev : NULL;
> +
>      __inc_zone_page_state(page, NR_FILE_DIRTY);
>      __inc_bdi_stat(mapping->backing_dev_info,
>        BDI_RECLAIMABLE);
>      task_io_account_write(PAGE_CACHE_SIZE);
> +    cgroup_io_acct = PAGE_CACHE_SIZE;
>     }
>     radix_tree_tag_set(&mapping->page_tree,
>      page_index(page), PAGECACHE_TAG_DIRTY);
> @@ -1100,6 +1108,7 @@ int __set_page_dirty_nobuffers(struct page *page)
>     /* !PageAnon && !swapper_space */
>     __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
>    }
> +  cgroup_io_throttle(bdev, cgroup_io_acct);
>    return 1;
>   }
>   return 0;
> diff --git a/mm/readahead.c b/mm/readahead.c
> index d8723a5..dff6b02 100644
> --- a/mm/readahead.c
> +++ b/mm/readahead.c
> @@ -14,6 +14,7 @@
>  #include <linux/blkdev.h>
>  #include <linux/backing-dev.h>
>  #include <linux/task_io_accounting_ops.h>
> +#include <linux/blk-io-throttle.h>
>  #include <linux/pagevec.h>
>  #include <linux/pagemap.h>
>
```

> @@ -58,6 +59,9 @@ int read_cache_pages(struct address_space *mapping, struct list_head *pages,
> int (*filler)(void *, struct page *), void *data)
> {
> struct page *page;
> + struct block_device *bdev =
> + (mapping->host && mapping->host->i_sb->s_bdev) ?
> + mapping->host->i_sb->s_bdev : NULL;
> int ret = 0;
>
> while (!list_empty(pages)) {
> @@ -76,6 +80,7 @@ int read_cache_pages(struct address_space *mapping, struct list_head *pages,
> break;
> }
> task_io_account_read(PAGE_CACHE_SIZE);
> + cgroup_io_throttle(bdev, PAGE_CACHE_SIZE);
> }
> return ret;
> }

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 3/3] i/o accounting and control
Posted by Andrea Righi on Thu, 26 Jun 2008 22:37:25 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Fri, 20 Jun 2008 12:05:35 +0200
> Andrea Righi <righi.andrea@gmail.com> wrote:
>
>> Apply the io-throttle controller to the opportune kernel functions. Both
>> accounting and throttling functionalities are performed by
>> cgroup_io_throttle().
>>
>> Signed-off-by: Andrea Righi <righi.andrea@gmail.com>
>> ---
>> block/blk-core.c | 2 ++
>> fs/buffer.c | 10 ++++++++++
>> fs/direct-io.c | 3 +++
>> mm/page-writeback.c | 9 +++++++++
>> mm/readahead.c | 5 +++++
>> 5 files changed, 29 insertions(+), 0 deletions(-)
>>

>> diff --git a/block/blk-core.c b/block/blk-core.c
>> index 1905aab..8eddef5 100644
>> --- a/block/blk-core.c
>> +++ b/block/blk-core.c
>> @@ -26,6 +26,7 @@
>> #include <linux/swap.h>
>> #include <linux/writeback.h>
>> #include <linux/task_io_accounting_ops.h>
>> +#include <linux/blk-io-throttle.h>
>> #include <linux/interrupt.h>
>> #include <linux/cpu.h>
>> #include <linux/blktrace_api.h>
>> @@ -1482,6 +1483,7 @@ void submit_bio(int rw, struct bio *bio)
>>   count_vm_events(PGPGOUT, count);
>>   } else {
>>   task_io_account_read(bio->bi_size);
>> +   cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
>
> We do this on every submit_bio(READ)? Ow.
>
> I guess it's not _hugely_ expensive, but the lengthy pointer hop in
> task_subsys_state() is going to cost.
>
> All this could be made cheaper if we were to reduce the sampling rate:
> only call cgroup_io_throttle() on each megabyte of IO (for example).
>
> current->amount_of_io += bio->bi_size;
> if (current->amount_of_io > 1024*1024) {
>   cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
>   current->amount_of_io -= 1024 * 1024;
> }

What about ratelimiting the sampling based on i/o requests?

```
 current->io_requests++;
 if (current->io_requests > CGROUP_IOTHROTTLE_RATELIMIT) {
  cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
  current->io_requests = 0;
 }
```

The throttle would fail for large bio->bi_size requests, but it would
work also with multiple devices. And probably this would penalize tasks
having a seeky i/o workload (many requests means more checks for
throttling).

>
> but perhaps that has the potential to fail to throttle correctly when
> accessing multiple devices, dunno.

>
>
> Bear in mind that tasks such as pdflush and kswapd will do reads when
> performing writeout (indirect blocks).
>
>>   count_vm_events(PGPGIN, count);
>>   }
>>
>> diff --git a/fs/buffer.c b/fs/buffer.c
>> index a073f3f..c63dfe7 100644
>> --- a/fs/buffer.c
>> +++ b/fs/buffer.c
>> @@ -35,6 +35,7 @@
>> #include <linux/suspend.h>
>> #include <linux/buffer_head.h>
>> #include <linux/task_io_accounting_ops.h>
>> +#include <linux/blk-io-throttle.h>
>> #include <linux/bio.h>
>> #include <linux/notifier.h>
>> #include <linux/cpu.h>
>> @@ -700,6 +701,9 @@ EXPORT_SYMBOL(mark_buffer_dirty_inode);
>> static int __set_page_dirty(struct page *page,
>>   struct address_space *mapping, int warn)
>> {
>> + struct block_device *bdev = NULL;
>> + size_t cgroup_io_acct = 0;
>> +
>>   if (unlikely(!mapping))
>>   return !TestSetPageDirty(page);
>>
>> @@ -711,16 +715,22 @@ static int __set_page_dirty(struct page *page,
>>   WARN_ON_ONCE(warn && !PageUptodate(page));
>>
>>   if (mapping_cap_account_dirty(mapping)) {
>> +   bdev = (mapping->host &&
>> +   mapping->host->i_sb->s_bdev) ?
>> +   mapping->host->i_sb->s_bdev : NULL;
>> +
>>   __inc_zone_page_state(page, NR_FILE_DIRTY);
>>   __inc_bdi_stat(mapping->backing_dev_info,
>>   BDI_RECLAIMABLE);
>>   task_io_account_write(PAGE_CACHE_SIZE);
>> +   cgroup_io_acct = PAGE_CACHE_SIZE;
>>   }
>>   radix_tree_tag_set(&mapping->page_tree,
>>   page_index(page), PAGECACHE_TAG_DIRTY);
>>   }
>>   write_unlock_irq(&mapping->tree_lock);

>> __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
>> + cgroup_io_throttle(bdev, cgroup_io_acct);
>
> Ah, and here we see the reason for the __cant_sleep() hack.
>
> It doesn't work, sorry. in_atomic() will return false when called
> under spinlock when CONFIG_PREEMPT=n. Your code will call
> schedule_timeout_killable() under spinlock and is deadlockable.
>
> We need to think of something smarter here. This problem has already
> been largely solved at the balance_dirty_pages() level. Perhaps that
> is where the io-throttling should be cutting in?
>
>> return 1;
>> }
>> diff --git a/fs/direct-io.c b/fs/direct-io.c
>> index 9e81add..fe991ac 100644
>> --- a/fs/direct-io.c
>> +++ b/fs/direct-io.c
>> @@ -35,6 +35,7 @@
>> #include <linux/buffer_head.h>
>> #include <linux/rwsem.h>
>> #include <linux/uio.h>
>> +#include <linux/blk-io-throttle.h>
>> #include <asm/atomic.h>
>>
>> /*
>> @@ -666,6 +667,8 @@ submit_page_section(struct dio *dio, struct page *page,
>>   /*
>>   * Read accounting is performed in submit_bio()
>>   */
>> + struct block_device *bdev = dio->bio ? dio->bio->bi_bdev : NULL;
>> + cgroup_io_throttle(bdev, len);
>>   task_io_account_write(len);
>> }
>>
>> diff --git a/mm/page-writeback.c b/mm/page-writeback.c
>> index 789b6ad..a2b820d 100644
>> --- a/mm/page-writeback.c
>> +++ b/mm/page-writeback.c
>> @@ -23,6 +23,7 @@
>> #include <linux/init.h>
>> #include <linux/backing-dev.h>
>> #include <linux/task_io_accounting_ops.h>
>> +#include <linux/blk-io-throttle.h>
>> #include <linux/blkdev.h>
>> #include <linux/mpage.h>
>> #include <linux/rmap.h>

```
>> @@ -1077,6 +1078,8 @@ int __set_page_dirty_nobuffers(struct page *page)
>>   if (!TestSetPageDirty(page)) {
>>    struct address_space *mapping = page_mapping(page);
>>    struct address_space *mapping2;
>> +  struct block_device *bdev = NULL;
>> +  size_t cgroup_io_acct = 0;
>>
>>    if (!mapping)
>>     return 1;
>> @@ -1087,10 +1090,15 @@ int __set_page_dirty_nobuffers(struct page *page)
>>     BUG_ON(mapping2 != mapping);
>>     WARN_ON_ONCE(!PagePrivate(page) && !PageUptodate(page));
>>     if (mapping_cap_account_dirty(mapping)) {
>> +    bdev = (mapping->host &&
>> +     mapping->host->i_sb->s_bdev) ?
>> +     mapping->host->i_sb->s_bdev : NULL;
>> +
>>      __inc_zone_page_state(page, NR_FILE_DIRTY);
>>      __inc_bdi_stat(mapping->backing_dev_info,
>>        BDI_RECLAIMABLE);
>>      task_io_account_write(PAGE_CACHE_SIZE);
>> +    cgroup_io_acct = PAGE_CACHE_SIZE;
>>     }
>>     radix_tree_tag_set(&mapping->page_tree,
>>      page_index(page), PAGECACHE_TAG_DIRTY);
>> @@ -1100,6 +1108,7 @@ int __set_page_dirty_nobuffers(struct page *page)
>>     /* !PageAnon && !swapper_space */
>>     __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
>>    }
>> +  cgroup_io_throttle(bdev, cgroup_io_acct);
>>    return 1;
>>   }
>>   return 0;
>> diff --git a/mm/readahead.c b/mm/readahead.c
>> index d8723a5..dff6b02 100644
>> --- a/mm/readahead.c
>> +++ b/mm/readahead.c
>> @@ -14,6 +14,7 @@
>>  #include <linux/blkdev.h>
>>  #include <linux/backing-dev.h>
>>  #include <linux/task_io_accounting_ops.h>
>> +#include <linux/blk-io-throttle.h>
>>  #include <linux/pagevec.h>
>>  #include <linux/pagemap.h>
>>
>> @@ -58,6 +59,9 @@ int read_cache_pages(struct address_space *mapping, struct list_head
*pages,
>>     int (*filler)(void *, struct page *), void *data)
```

>> {
>>   struct page *page;
>> + struct block_device *bdev =
>> + (mapping->host && mapping->host->i_sb->s_bdev) ?
>> + mapping->host->i_sb->s_bdev : NULL;
>>   int ret = 0;
>>
>>   while (!list_empty(pages)) {
>> @@ -76,6 +80,7 @@ int read_cache_pages(struct address_space *mapping, struct list_head
*pages,
>>   break;
>>   }
>>   task_io_account_read(PAGE_CACHE_SIZE);
>> + cgroup_io_throttle(bdev, PAGE_CACHE_SIZE);
>>   }
>>   return ret;
>> }

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 3/3] i/o accounting and control
Posted by akpm on Thu, 26 Jun 2008 23:15:50 GMT
View Forum Message <> Reply to Message

On Fri, 27 Jun 2008 00:37:25 +0200
Andrea Righi <righi.andrea@gmail.com> wrote:

> > All this could be made cheaper if we were to reduce the sampling rate:
> > only call cgroup_io_throttle() on each megabyte of IO (for example).
> >
> > current->amount_of_io += bio->bi_size;
> > if (current->amount_of_io > 1024*1024) {
> >   cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
> >   current->amount_of_io -= 1024 * 1024;
> > }
>
> What about ratelimiting the sampling based on i/o requests?
>
>   current->io_requests++;
>   if (current->io_requests > CGROUP_IOTHROTTLE_RATELIMIT) {
>   cgroup_io_throttle(bio->bi_bdev, bio->bi_size);
>   current->io_requests = 0;
>   }
>
> The throttle would fail for large bio->bi_size requests, but it would

> work also with multiple devices. And probably this would penalize tasks
> having a seeky i/o workload (many requests means more checks for
> throttling).

Yup. To a large degree, a 4k IO has a similar cost to a 1MB IO.
Certainly the 1MB IO is not 256 times as expensive!

Some sort of averaging fudge factor could be used here. For example, a
1MB IO might be considered, umm 3.4 times as expensive as a 4k IO. But
it varies a lot depending upon the underlying device. For a USB stick,
sure, we're close to 256x. For a slow-seek, high-bandwidth device
(optical?) it's getting closer to 1x. No single fudge-factor will suit
all devices, hence userspace-controlled tunability would be needed here
to avoid orders-of-magnitude inaccuracies.

The above

 cost ~= per-device-fudge-factor * io-size

can of course go horridly wrong because it doesn't account for seeks at
all. Some heuristic which incorporates per-cgroup seekiness (in some
weighted fashion) will help there.

I dunno. It's not a trivial problem, and I suspect that we'll need to
get very fancy in doing this if we are to avoid an implementation which
goes unusably badly wrong in some situations.

I wonder if we'd be better off with a completely different design.

<thinks for five seconds>

At present we're proposing that we look at the request stream and
a-priori predict how expensive it will be. That's hard. What if we
were to look at the requests post-facto and work out how expensive they
were? Say, for each request which this cgroup issued, look at the
start-time and the completion-time. Take the difference (elapsed time)
and divide that by wall time. Then we end up with a simple percentage:
"this cgroup is using the disk 10% of the time".

That's fine, as long as nobody else is using the device! If multiple
cgroups are using the device then we need to do, err, something.

Or do we? Maybe not - things will, on average, sort themselves out,
because everyone will slow everyone else down. It'll have inaccuracies
and failure scenarios and the smarty-pants IO schedulers will get in

the way.

Interesting project, but I do suspect that we'll need a lot of
complexity in there (and huge amounts of testing) to get something
which is sufficiently accurate to be generally useful.

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers