Request-based Device-mapper multipath and Dynamic load balancing

Kiyoshi Ueda
NEC Corporation
`k-ueda@ct.jp.nec.com`

Jun'ichi Nomura
NEC Corporation
`j-nomura@ce.jp.nec.com`

Mike Christie
Red Hat, Inc.
`mchristi@redhat.com`

Abstract

Multipath I/O is the ability to provide increased performance and fault tolerant access to a device by addressing it through more than one path. For storage devices, Linux has seen several solutions that were of two types: high-level approaches that live above the I/O scheduler (BIO mappers), and low-level subsystem specific approaches. Each type of implementation has its advantage because of the position in the storage stack in which it has been implemented. The authors focus on a solution that attempts to reap the benefits of each type of solution by moving the kernel's current multipath layer, `dm-multipath`, below the I/O scheduler and above the hardware-specific subsystem. This paper describes the block, Device-mapper, and SCSI layer changes for the solution and its effect on performance.

1 Introduction

Multipath I/O provides the capability to utilize multiple paths between the host and the storage device. Multiple paths can result from host or storage controllers having more than one port, redundancy in the fabric, or having multiple controllers or buses.

As can be seen in Figure 1(a), multipath architectures like MD Multipath or the Device-mapper (DM) layer's multipath target have taken advantage of multiple paths at a high level by creating a virtual device that is comprised of block devices created by the hardware subsystem.

In this approach, the hardware subsystem is unaware of multipath, and the lower-level block devices represent paths to the storage device which the higher-level software routes I/Os in units of BIOs over [1]. This design has the benefit that it can support any block device, and is easy to implement because it uses the same infrastructure that software RAID uses. It has the drawback that it does not have access to detailed error information, and it resides above the I/O scheduler and I/O merging, which makes it difficult to make load-balancing decisions.

Another approach to multipath design modifies the hardware subsystem or low-level driver (LLD) to be multipath-aware. For the Linux SCSI stack (Figure 1(b)), Host Bus Adapter (HBA) manufacturers, such as Qlogic, have provided LLDs for their Fibre Channel and iSCSI cards which hide the multipath details from the OS [2]. These drivers coalesce paths into a single device which is exposed to the kernel, and route SCSI commands or driver-specific structures. There have also been multipath implementations in the SCSI layer that are able to route SCSI commands over different types of HBAs. These can be implemented above the LLD in the SCSI mid-layer or as a specialized SCSI upper-layer driver [3]. These designs benefit from being at a lower level, because they are able to quickly distinguish transport problems from device errors, can support any SCSI device including tape, and are able to make more intelligent load-balancing decisions.

This paper describes a multipath technique, ***Request***-based DM, which can be seen in Figure 1(c), that is located under the I/O scheduler and above the hardware subsystem, and routes `struct request`s over the devices created by the hardware specific subsystem. (To distinguish from the general meaning of *request*, ***request*** is used in this paper to mention the `struct request`.) As a result of working with ***request***s instead of BIOs or SCSI commands, this new technique is able to bridge the multipath layer with the lower levels because it has access to the lower-level error information, and is able to leverage the existing block-layer statistics and queue management infrastructure to provide improved load balancing.

In Section 2, we will give an overview of the Linux block and DM layer. Then Section 3 describes in more detail the differences between routing I/O at the BIO-level versus the ***request***-level, and what modifications to

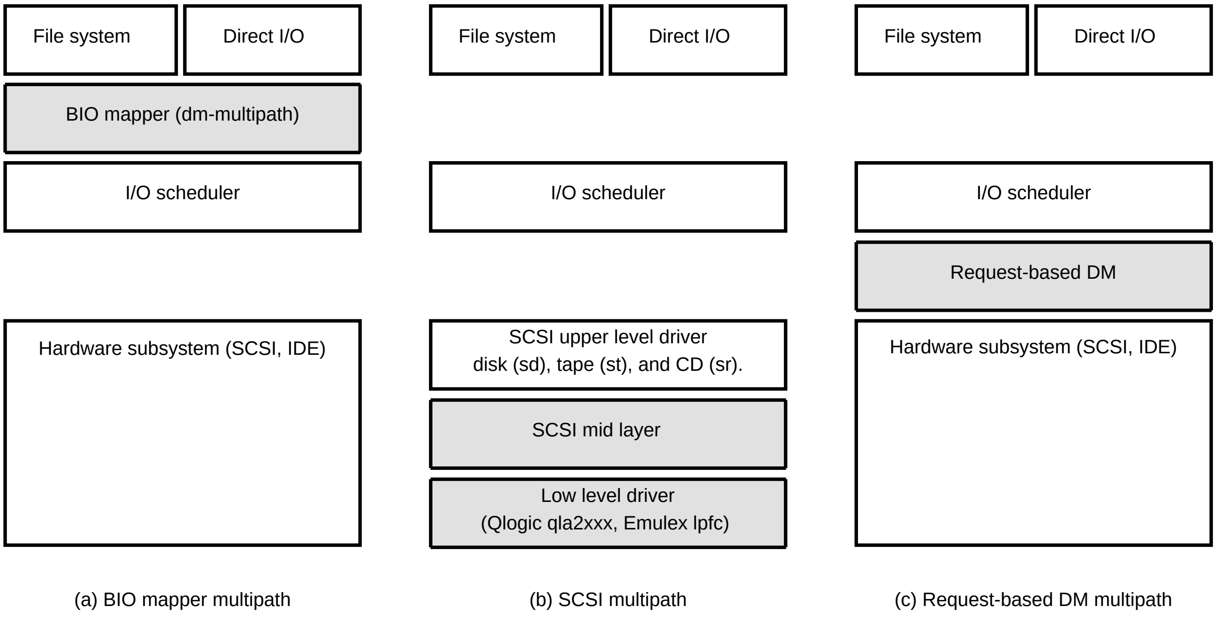


Figure 1: Multipath implementations

the block, SCSI, and DM layers are necessary to support ***Request***-based multipath. Finally, we will detail performance results and load-balancing changes in Section 4 and Section 5.

2 Overview of Linux Block I/O and DM

In this section, we look into the building blocks of multipath I/O, specifically, how the applications' read/write operations are translated into low-level I/O requests.

2.1 Block I/O

When a process, or kernel itself, reads or writes to a block device, the I/O operation is initially packed into a simple container called a BIO. (See Figure 2.) The BIO uses a vector representation pointing to an array of tuples of `<page, offset, len>` to describe the I/O buffer, and has various other fields describing I/O parameters and state that needs to be maintained for performing the I/O [4].

Upon receiving the BIO, the block layer attempts to merge it with other BIOs already packed into ***request***s and inserted in the ***request*** queue. This allows the block layer to create larger ***request***s and to collect enough of them in the queue to be able to take advantage of the sorting/merging logic in the elevator [4].

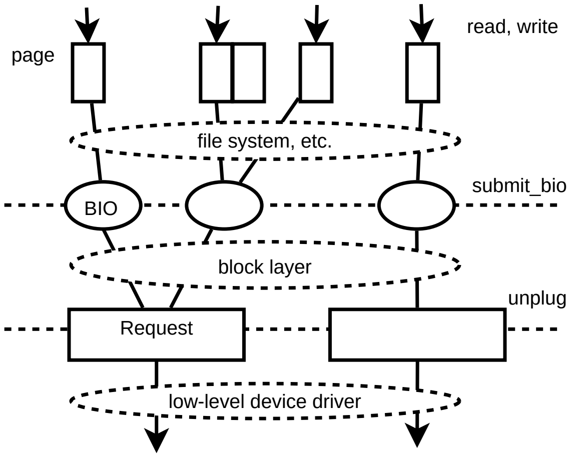


Figure 2: Relationship between page, BIO, and ***request***

This approach to collecting multiple larger ***request***s before dequeueing is called plugging. If the buffers of two BIOs are contiguous on disk, and the size of the BIO combined with the ***request*** it is to be merged with is within the LLD and hardware's I/O size limitations, then the BIO is merged with an existing ***request***; otherwise the BIO is packed into a new ***request*** and inserted into the block device's ***request*** queue. ***Request***s will continue to be merged with incoming BIOs and adjacent ***request***s until the queue is unplugged. An unplug of the queue is triggered by various events such as the plug timer firing, the number of ***request***s exceeding a given threshold, and I/O completion.

When the queue is unplugged the ***request***s are passed to the low-level driver's `request_fn()` to be executed. And when the I/O completes, subsystems like the SCSI and IDE layer will call `blk_complete_request()` to schedule further processing of the ***request*** from the block layer softirq handler. From that context, the subsystem will complete processing of the ***request*** by asking the block layer to requeue it, or by calling `end_request()`, or `end_that_request_first()/chunk()` and `end_that_request_last()` to pass ownership of the ***request*** back to the block layer. To notify upper layers of the completion of the I/O, the block layer will then call each BIO's `bi_end_io()` function and the ***request***'s `end_io()` function.

During this process of I/O being merged, queued and executed, the block layer and hardware specific subsystem collect a wide range of I/O statistics such as: the number of sectors read/written, the number of merges occurred and accumulated length of time ***request***s took to complete.

2.2 DM

DM is a virtual block device driver that provides a highly modular kernel framework for stacking block device filter drivers [1]. The filter drivers are called target drivers in DM terminology, and can map a BIO to multiple block devices, or can modify a BIO's data or simulate various device conditions and setups. DM performs this BIO manipulation by performing the following operations:

- Cloning
  - When a BIO is sent to the DM device, a near identical copy of that BIO is created by DM's `make_request()` function. The major difference between the two BIOs is that the clone will have a completion handler that points to DM's `clone_endio()` function.
- Mapping
  - The cloned BIO is passed to the target driver, where the clone's fields are modified. By targets that map or route I/O, the `bi_bdev` field is set to the device it wants to send the BIO to.
  - The cloned BIO is submitted to the underlying device.
- Completion
  - DM's completion handler calls the target-specific completion handler where the BIO can be remapped or completed.
  - Original BIO is completed when all cloned BIOs are completed.

2.3 `dm-multipath`

`dm-multipath` is the DM target driver that implements multipath I/O functionality for block devices. Its map function determines which device to route the BIO to using a path selection module and by ordering paths in priority groups (Figure 3). Currently, there is only the round-robin path selector, which sends *N* number of BIOs to a path before selecting a new path.

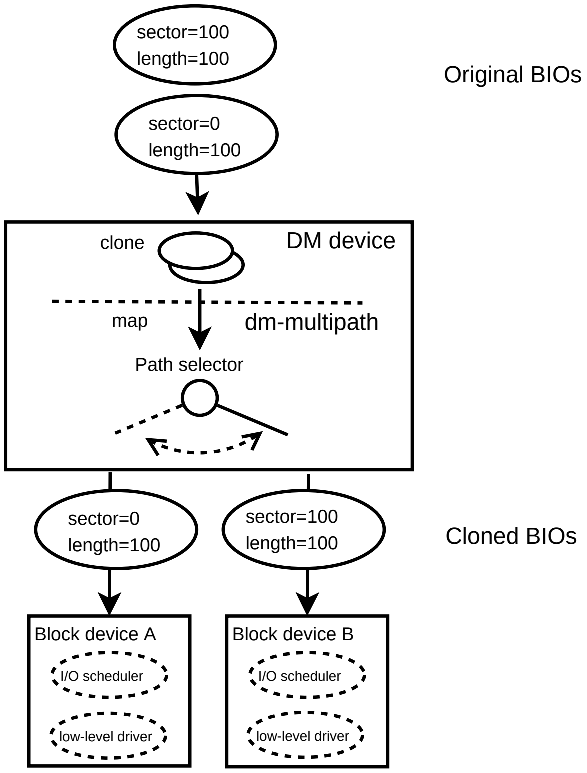


Figure 3: current (BIO-based) `dm-multipath`

3 *Request*-based DM

3.1 Limitations of BIO-based `dm-multipath`

There are three primary problems which ***Request***-based multipath attempts to overcome:

1. Path selection does not consider I/O scheduler's behavior.
2. Block layer statistics are difficult for DM to use.
3. Error severity information is not available to DM.

`dm-multipath`'s path selection modules must balance trying to plug the underlying ***request*** queue and creating large ***request***s against making sure each path is fully utilized. Setting the number of BIOs that cause a path switch to a high value will assure that most BIOs are merged. However, it can cause other paths to be underused because it concentrates I/O on a single path for too long. Setting the BIO path-switching threshold too low would cause small ***request***s to be sent down each path and would negate any benefits that plugging the queue would bring. At its current location in the storage stack, the path selector module must guess what will be merged by duplicating the block layer tests or duplicate the block layer statistics so it can attempt to make reasonable decisions based on past I/O trends.

Along with enhancing performance, multipath's other duty is better handling of disruptions. The LLD and the hardware subsystem have a detailed view of most problems. They can decode SCSI sense keys that may indicate a storage controller on the target is being maintained or is in trouble, and have access to lower level error information such as whether a connection has failed. Unfortunately, the only error information DM receives is the error code value `-EIO`.

Given that these issues are caused by the location of the dm-multipath map and completion functions, ***Request***-based DM adds new DM target drivers call outs, and modifies DM and the block layer to support mapping at the ***request***-level. Section 3.2 will explain the design and implementation of ***Request***-based DM. Section 3.3 discusses the issues that are still being worked on and problems that were discovered with the ***Request***-based approach. Finally, Section 3.4 describes user interface changes to DM.

3.2 Design and implementation

Request-based DM is based on ideas from the block layer multipath [5] patches, which were an experiment that moved the multipath layer down to the ***request***-level. The goal of ***Request***-based DM is to integrate into the DM framework so that it is able to utilize existing applications such as `multipath-tools` and `dmsetup`.

As explained in Section 2.2, DM's key I/O operations are cloning, mapping, and completion of BIOs. To allow these operations to be executed on ***request***s, DM was modified to take advantage of following block layer interfaces:

BIO	Submission	`make_request_fn`
	Completion	`bio->bi_end_io`
Request	Submission	`request_fn`
	Completion	`request->end_io`

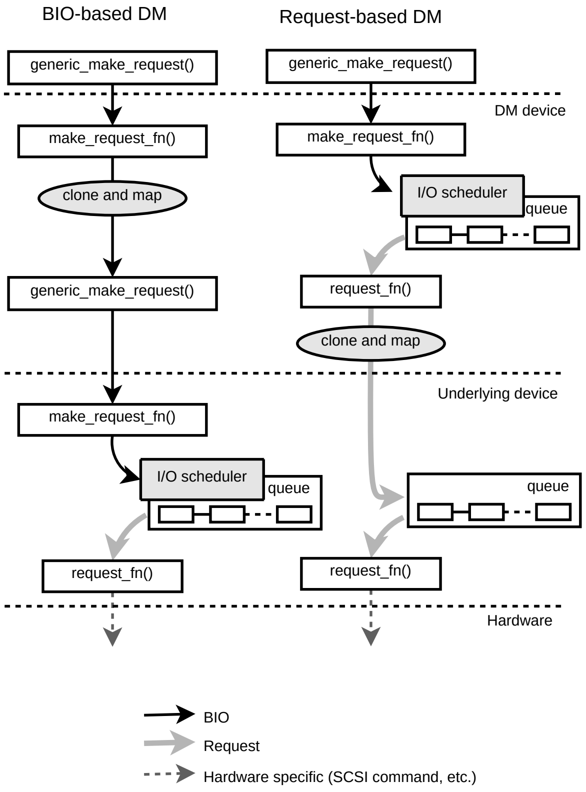


Figure 4: Difference of I/O submission flow

As can be seen in Figure 4, with BIO-based DM, BIOs submitted by upper layers through `generic_make_request()` are sent to DM's `make_request_fn()` where they are cloned, mapped, and then sent to a device below DM. This device could be another virtual device, or it could be the real device queue—in which case the BIO would be merged with a ***request*** or packed in a new one and queued, and later executed when the queue is unplugged.

Conversely, with ***Request***-based DM, DM no longer uses a specialized `make_request_fn()` and instead uses the default `make_request_fn()`, `__make_request()`. To perform cloning and mapping, DM now implements its `request_fn()` callback, which is called when the queue is unplugged. From this callback, the original ***request*** is cloned. The target driver is asked to map the clone. And then the clone is inserted directly into the underlying device's ***request*** queue using `__elv_add_request()`.

To handle I/O completion, the ***Request***-based DM patches allow the ***request***'s `end_io()` callback to be used to notify upper layers when an I/O is finished. More details on this implementation in Section 3.3. DM only needs to hook into the cloned ***request***'s completion handler—similar to what is done for BIO completions. (Figure 5).

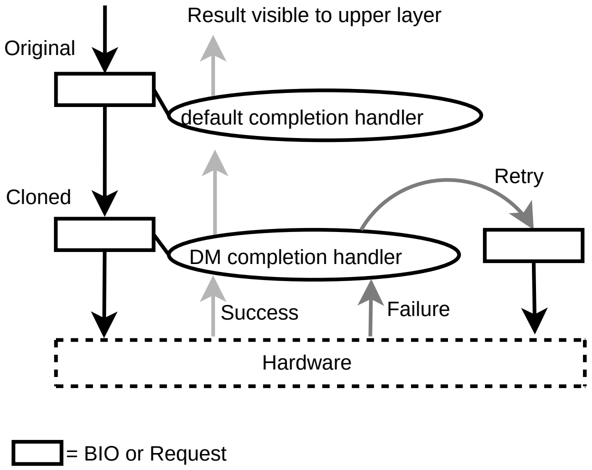


Figure 5: Completion handling of cloned I/O

3.3 *Request*-based DM road bumps

A lot of the ***Request***-based details have been glossed over, so they could be discussed in this section. While working with ***request***s simplifies the mapping operation, it complicates the cloning path, and requires a lot of new infrastructure for the completion path.

3.3.1 *Request* cloning

A clone of the original ***request*** is needed, because a layer below DM may modify ***request*** fields, and the use of a clone simplifies the handling of partial completions. To allocate the clone, it would be best to use the existing ***request*** mempools [6], so DM initially attempted to use `get_request()`. Unfortunately, the block layer and some I/O schedulers assume that `get_request()` is called in the process context in which the BIO was submitted, but a ***request*** queue's `request_fn()` can be called from the completion callback which can be run in a softirq context.

Removing the need for cloning or modifying `get_request()` to be usable from any context would be the preferable solutions, but both required too many changes to the block layer for the initial release. Instead, DM currently uses a private mempool for the ***request*** clones.

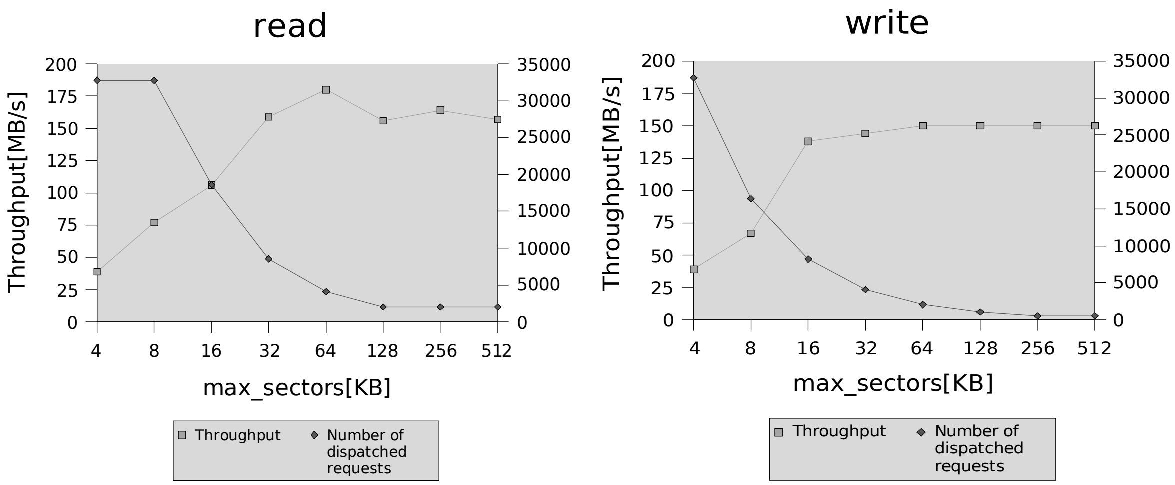
3.3.2 Completion handling

`dm-multipath`'s completion handler has to do the following:

- Check completion status.
- Setup a retry if an error has occurred.
- Release its private data if a retry is not needed.
- Return a result to the upper layer.

When segments of a BIO are completed, the upper layers do not begin processing the completion until the entire operation has finished. This allows BIO mappers to hook at a single point—the BIO's `bi_end_io()` callback. Before sending a BIO to `generic_make_request()`, DM will copy the mutable fields like the size, starting sector, and segment/vector index. If an error occurs, DM can wait until the entire I/O has completed, restore the fields it copied, and then retry the BIO from the beginning.

Requests, on the other hand, are completed in two parts: `__end_that_request_first()`, which completes BIOs in the ***request*** (sometimes partially), and `end_that_request_last()`, which handles statistics accounting, releases the ***request***, and calls its `end_`



Command: `dd if=/dev/<dev|zero> of=/dev/<null|dev> bs=16777216 count=8`

Figure 6: Performance effects of I/O merging

`io()` function. This separation creates a problem for error processing, because ***Request***-based DM does not do a deep copy of the ***request***'s BIOs. As a result, if there is an error, `__end_that_request_first()` will end up calling the BIO's `bi_end_io()` callback and returning the BIO to the upper layer, before DM has a chance to retry it.

A simple solution might be to add a new hook in `__end_that_request_first()`, where the new hook is called before a BIO is completed. DM would then be responsible for completing the BIO when it was ready. However, it just imposes additional complexity on DM because DM needs to split its completion handler into error checking and retrying as the latter still has to wait for `end_that_request_last()`. It is nothing more than a workaround for the lack of ***request*** stacking.

True *request* stacking

To solve the problem, a redesign of the block layer ***request*** completion interfaces, so that they function like BIO stacking, is necessary. To accomplish this, ***Request***-based DM implements the following:

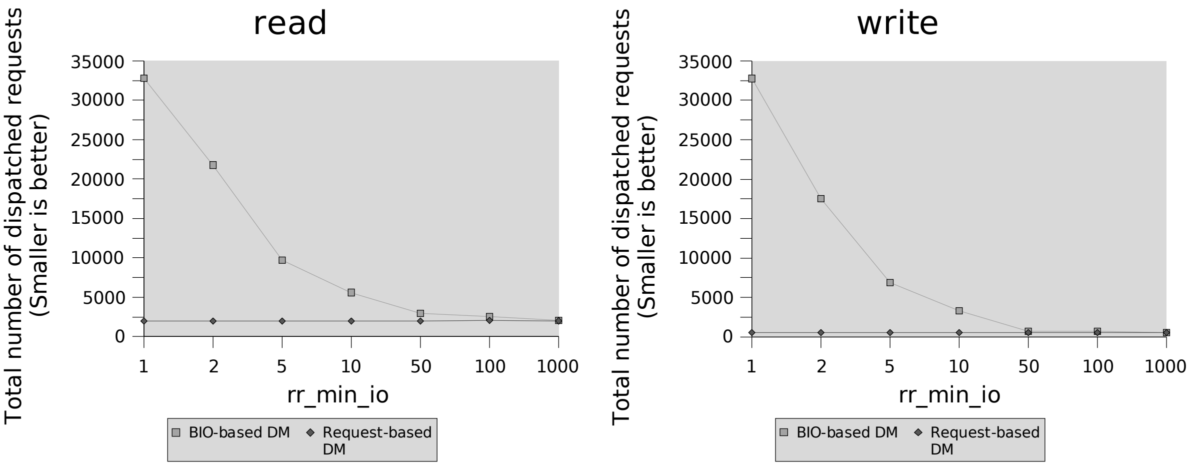
- Provide a generic completion handler for ***request***s which are not using their `end_io()` in the current code. `__make_request()` will set the default handler.

- Modify existing `end_io()` users to handle the new behavior.

- Provide a new helper function for device drivers to complete a ***request***. The function eventually calls the ***request***'s `end_io()` function. Device drivers have to be modified to call the new helper function for ***request*** completion. (`end_that_request_*()` are no longer called from device drivers.) If the helper function returns 0, device drivers can assume the ***request*** is completed. If the helper function returns 1, device drivers can assume the ***request*** is not completed and can take actions in response.

3.4 User-space DM interface

`dm-multipath` has a wide range of tools like `multipath-tools` and installers for major distributions. To minimize headaches caused by changing the multipath infrastructure, not only did ***Request***-based multipath hook into DM, but it also reused the BIO-based `dm-multipath` target. As a result, the only change to the user-space DM interface is a new flag for the DM device creation `ioctl`. The existence of the flag is checked at the ioctl handler and, if it is turned on, the device will be set up for ***Request***-based DM.



Command: `dd if=/dev/<dev|zero> of=/dev/<null|dev> bs=16777216 count=8`

Figure 7: Effects of frequent path changes on I/O merging

4 Performance testing

One of the main purposes of ***Request***-based `dm-multipath` is to reduce the total number of ***request***s dispatched to underlying devices even when path change occurs frequently.

In this section, performance effects of I/O merging and how ***Request***-based `dm-multipath` reduces the number of ***request***s under frequent path change are shown. The test environment used for the measurement is shown in Table 1.

Host	CPU	Intel Xeon 1.60[GHz]
	Memory	2[GB]
	FC HBA	Emulex LPe1150-F4 * 2
Storage	Port	4[Gbps] * 2
	Cache	4[GB]
Switch	Port	2[Gbps]

Table 1: Test environment

4.1 Effects of I/O merging

Sequential I/O results are shown in Figure 6. The throughput for a fixed amount of reads and writes on a local block device was measured using the `dd` command while changing the queue's `max_sectors` parameter.

In the test setup, the Emulex driver's max segment size and max segment count are set high enough, so that `max_sectors` controls the ***request*** size. It is expected that when the value of `max_sectors` becomes smaller, the number of dispatched ***request***s becomes larger. The results in Figure 6 appear to confirm this, and indicate that I/O merging, or at least larger I/Os, is important to achieve higher throughput.

4.2 Effects of *Request*-based `dm-multipath` on I/O merging

While the results shown in the previous section are obtained by artificially reducing the `max_sectors` parameter, such situations can happen when frequent path changes occur in BIO-based `dm-multipath`.

Testing results for the same sequential I/O pattern on a `dm-multipath` device when changing round-robin path selector's `rr_min_io` parameter which corresponds to the frequency of path change is shown in Figure 7.

This data shows that, when path change occurs frequently, the total number of ***request***s increases with BIO-based `dm-multipath`. While under the same condition, the number of ***request***s is low and stable with ***Request***-based `dm-multipath`.

4.3 Throughput of *Request*-based `dm-multipath`

At this point in time, ***Request***-based `dm-multipath` still cannot supersede BIO-based `dm-multipath` in sequential read performance with a simple round-robin

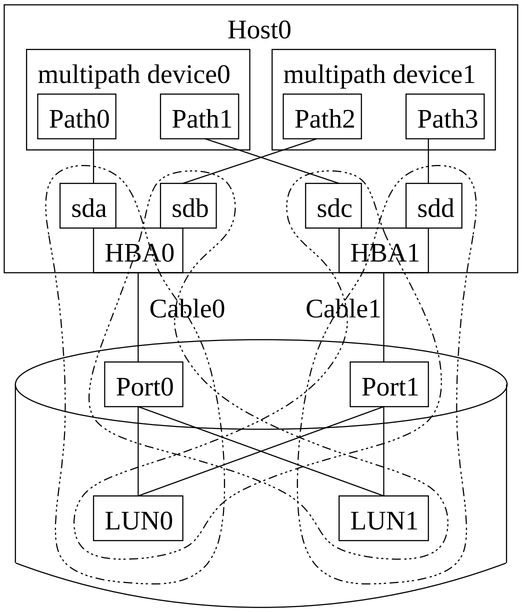


Figure 8: Examples of targets sharing cables

path selector. The performance problem is currently under investigation.

5 Dynamic load balancing

The benefit of moving multipath layer below the I/O scheduler is not only for the efficiency of I/O merging.

This section reviews the other important feature of ***Request***-based `dm-multipath`, dynamic load balancing.

5.1 Needs of dynamic load balancing

There are two load balancing types in general, static and dynamic. `dm-multipath` currently supports a static balancer with weighted round-robin. It may be sufficient in an ideal environment where all paths and storage controllers are symmetric and private to the host system. But in an environment where the load of each path is not same or dynamically changes, round-robin does not work well.

For example, suppose there is a multipath configuration described in Figure 8 and `Path0(sda)` is being used heavily for `multipath device0`. It means `Cable0` is heavily loaded. In this situation, `multipath device1` should use `Path3(sdd)` to avoid heavily loaded `Cable0`. However, the round-robin path selector does not care about that, and will select `Path2(sdb)` at next path selection for `multipath device1`. It will cause congestion on `Cable0`.

To get better performance even in such environments, a dynamic load balancer is needed.

5.2 Load parameters

It is important to define good metrics to model the load of a path. Below is an example of parameters which determine the load of a path.

- Number of in-flight I/Os;
- Block size of in-flight I/Os;
- Recent throughput;
- Recent latency; and
- Hardware specific parameters.

Using information such as the number of in-flight I/Os on a path would be the simplest way to gauge traffic. For BIO-based DM, it was described in Section 2.2 the unit of I/O is the BIO, but BIO counters do not take into account merges. ***Request***-based DM, on the other hand, is better suited for this type of measurement. Its use of ***request***s allows it to measure traffic in the same units of I/O that are used by lower layers, so it is possible for ***Request***-based DM to take into account a lower layer's limitations like HBA, device, or transport queue depths.

Throughput or latency is another valuable metric. Many lower layer limits are not dynamic, and even if they were, could not completely account for every bottleneck in the topology. If we used throughput as the load of a path, path selection could be done with the following steps.

1. Track block size of in-flight I/Os on each path . . . *in-flight size*.
2. At path selection time, calculate recent throughput of each path by using generic diskstats (`sectors` and `io_ticks`) . . . *recent throughput*.
3. For each path, calculate the time which all in-flight I/Os will finish by using (*in-flight size*)/ (*recent throughput*).

4. Select the path of which Step 3 is the shortest time.

We plan to implement such dynamic load balancers after resolving the performance problems of Section 4.3.

6 Conclusion

Request-based multipath has some potential improvements over current `dm-multipath`. The paper focused on the I/O merging, which affects load balancing, and confirmed the code works correctly.

There is a lot of work to be done to modify the block layer so that it can efficiently and elegantly handle routing ***request***s. And there are a lot of interesting directions the path selection modules can take, because the multipath layer is now working in the same units of I/O that the storage device and LLD are.

References

[1] Edward Goggin, Linux Multipathing *Proceedings of the Linux Symposium*, 2005.

[2] Qlogic, Qlogic QL4xxx QL2xxx Driver README, `http://www.qlogic.com`.

[3] Mike Anderson, SCSI Mid-Level Multipath, *Proceedings of the Linux Symposium*, 2003.

[4] Jens Axboe, Notes on the Generic Block Layer Rewrite in Linux 2.5, *Documentation/block/ biodoc.txt*, 2007.

[5] Mike Christie, [PATCH RFC] block layer (request based) multipath, `http://lwn.net/Articles/156058/`.

[6] Jonathan Corbet, Driver porting: low-level memory allocation, `http://lwn.net/Articles/22909/`.

Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*