Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

SCSI Fault Injection Test

Kenichi Tanaka
NEC Corporation
k-tanaka@ce.jp.nec.com

Masayuki Hamaguchi
NEC Software Tohoku, Ltd.
m-hamaguchi@ys.jp.nec.com

Takatoshi Sato
NEC Software Tohoku, Ltd.
t-sato@wm.jp.nec.com

Kosuke Tatsukawa
NEC Corporation
tatsu@ab.jp.nec.com

Abstract

It has been widely recognized that the testing of Linux kernel is important. However, error handling code is one of the places where testing is difficult. In this paper, a new block I/O test tool is introduced. This makes testing of error handling codes easy. The new test tool has driver level fault injectors which have flexible and fully controllable interface for user level programs to simulate real device errors. This paper describes the detailed design of the new test tool and a fault injector implementation for SCSI. Also, the usefulness of the new test tool is shown by actual evaluation of Linux software RAID drivers.

1 Introduction

There is an increasing opportunity to use Linux in enterprise systems, where the users expect very high reliability. Storage is one of the areas for which the highest reliability is required because its failure may cause system downtime and data loss. For the case of hardware failures, the operating system must provide high quality error handling. That means thorough testing of the error handling code is inevitable.

However, error handling code is one of the places where testing is difficult. There are two reasons why evaluation of error handling code is difficult;

- Error handling code is rarely executed. It can not be tested just by running the system under normal operation.
- Fault patterns vary. They can occur during various timings, and it is difficult to thoroughly test each combination.

Fault injection is a generally used technique to overcome the difficulty by controlling the fault occurrence and forcing the execution of error handling code. Several fault injection methods are already available for Linux but all of them lack either variety of fault patterns or flexibility to inject faults as intended, which are needed for systematic evaluation of error handling code. For example, with the existing fault injection methods, it is difficult to make a test program which tests error handling code of a hard disk drive (HDD) access timeout while the software RAID recovery is in progress.

In this paper, a new test tool with SCSI fault injection is introduced. The test tool is capable of injecting SCSI faults with realistic fault patterns and includes a set of test programs to cover various combinations of fault conditions. Since SCSI is the most widely used storage subsystem in Linux, this test tool enables systematic evaluation of error handlings in Linux block device drivers.

In section 2, design overview of the test tool and comparison with other existing fault injection tools are described. Design and implementation details are explained in section 3. Section 4 shows an example of evaluation using the test tool for Linux software RAID drivers, that was the original motivation of developing this test tool, and the result of the evaluation. We conclude in section 5 and explain possible future works.

2 Testing Error Handler Using SCSI Fault Injection

In order to systematically test error handling code using fault injection for a target kernel component which is being tested, a set of test programs is necessary where each individual test program checks whether a certain type of fault occurring when the target kernel component is in a certain state is handled correctly. It is necessary for the test program to prepare the target kernel module to be in the desired state, and then inject the desired fault on the desired access which the test program will trigger, and test if the result is correct.

In order to achieve this goal, the fault injector provides an interface to specify the type of fault which will be injected, and on which access will cause the injection.

2.1 Specifying the Type of Fault

The SCSI HDD fault patterns will be categorized to determine the fault pattern which the fault injector has to generate.

SCSI fault can be classified in two patterns. One is "The SCSI device respond with an error" pattern, which is the case when the drive explicitly returns an error condition to the OS. The other is "The device does not respond" pattern, which is the case when the drive does not return any status to the OS resulting in a timeout. For example, the former can be caused by media error and the latter can be caused by SCSI cable fault.

Alternatively, HDD hardware fault can be divided into temporary faults and permanent faults. A temporary fault can be caused by an accidental and recoverable HDD fault. A permanent fault can be caused by a severe HDD fault.

Based on the type of access which will cause the fault in each of the above four areas, we have categorized the HDD faults into the following eight categories.

Temporary faults with error return can be classified into the following two cases, based on the type of access.

1. Temporary read error – This type of fault is accidentally caused by read access, which occurs just once.

2. Temporary write error – This type of fault is accidentally caused by write access, which occurs just once.

Permanent error with error return can be classified into the following three cases.

3. Read error correctable by write – This type of fault is a medium error which can be corrected by writing data to the failed sector. After writing to the sector, subsequent reads and writes will both succeed.

4. Permanent read error – This type of fault is a permanent medium error on a particular sector. Any read access to the sector fails, but write will succeed, because many disks can not detect errors while writing data to the medium.

5. Permanent read/write error – This type of fault is a severe error. Both read and write fail permanently.

Temporary timeout errors can be classified into the following two cases, based on the type of request.

6. Temporary no response on read access – This type of fault can be caused by congestion, resulting in SCSI command timeout on a read access. After the congestion disappears, both read and write accesses will succeed.

7. Temporary no response on write access – This type of fault can be caused by congestion, resulting in SCSI command timeout on a write access. After the congestion disappears, both read and write accesses will succeed.

Practically, permanent timeout errors occur regardless of the type of request, read or write. So we have only one class for this type of error.

8. Permanent no response on either read or write access – This type of fault is a device failure resulting in SCSI command timeout. Both read and write requests fail permanently.

2.2 Specifying the Access to Trigger the Fault

Fault location of a SCSI HDD can be identified by the disk device and the failed sector within the disk. In Linux, the disk device can be specified by the major number and minor number of the block device.

The failed sector can be specified by the sector number. However it is difficult for a user-level test program to be aware of the sector number. So, the fault injector also accepts either the file system block number or the inode number to specify the fault location. Those numbers will be automatically converted to the corresponding sector number by the fault injector.

2.3 Design Comparison with Existing Methods

Several methods have already been proposed for injecting faults into block I/O processing, which can be used for evaluation of block I/O error handling code;

- *Linux `scsi_debug` driver* – This is a SCSI low level driver used for debugging. This driver creates a simulated SCSI device using a ramdisk and is capable of injecting various SCSI faults when accessed. However, the condition for injecting faults is limited. For example medium error can only be injected by accessing sector 0x1234 [6].
- *Linux Fault Injection Framework* – This kernel feature, which was merged into 2.6.20 kernel, is used to inject various types of errors into Linux kernel. The framework also supports I/O fault injection, but the fault pattern it can simulate is very limited. For example, it can not inject faults to simulate device timeouts.
- *Using special hardware* – This method uses special hardware for fault injection such as failed HDD. The most precise evaluation result can be obtained since actual hardware is used to inject faults. However, the availability of such hardware is very limited and they are typically expensive.

These existing fault injection methods do not have enough flexibility to inject various faults into the system as intended, which is needed for systematic evaluation of error handling code. The proposed SCSI fault injector described in this paper has a following benefits compared with existing methods.

- *A flexible fault injection trigger* – The SCSI fault injector can trigger a fault on accessing the user specified location of any SCSI device, which is missing in the `scsi_debug` driver.
- *A realistic fault simulation* – The SCSI fault injector can simulate a realistic fault condition by inserting a fault generation code in the SCSI driver, which is missing in the *Linux Fault Injection Framework*.
- *No needs for external hardware* – The SCSI fault injector provides a realistic fault simulation without any external hardware. Also it does not require software modification including Linux kernel.

3 SCSI Fault Injector

In this section, the design and implementation of the SCSI fault injector is explained in detail.

The SCSI fault injector is implemented as a set of SystemTap scripts. SystemTap is used to track information when an I/O request is passed between various layers within the kernel, and to add a hook to inject a fault.

SystemTap provides infrastructure to embed a hook in the kernel dynamically, and to change the value of variables or function return values. Also, SystemTap makes it possible to keep these values in SystemTap variables. By using SystemTap, a fake response from a SCSI device can be created as if a SCSI device had reported an error to the OS [7].

The SCSI fault injector takes a fault pattern and a trigger condition as arguments from a test program. Once started, the injector tracks I/O requests and injects a fault if the condition is met.

The fault injection works in 2 steps.

1. Identify the target SCSI command matching the user-specified condition
2. Inject a fault in the processing of the target SCSI command

3.1 Identifying the target SCSI command

The flow of block I/O processing from a user space test program to the SCSI middle layer is described. Also it is explained how the SCSI fault injector tracks the request to identify the target SCSI command in the flow.

A test program can initiate an I/O request with a read or write system call to the Linux kernel (see Figure 1).

The system call is sent to filesystem layer and eventually translated into a `struct bio`, which is passed to the block I/O layer by the `submit_bio()` function.

The `bio` contains the necessary information for performing I/O. Especially, a `bio` has `bi_sector` and `bi_size`, which represent the logical I/O destination of the target block device and access length respectively at the beginning of `submit_bio()`.

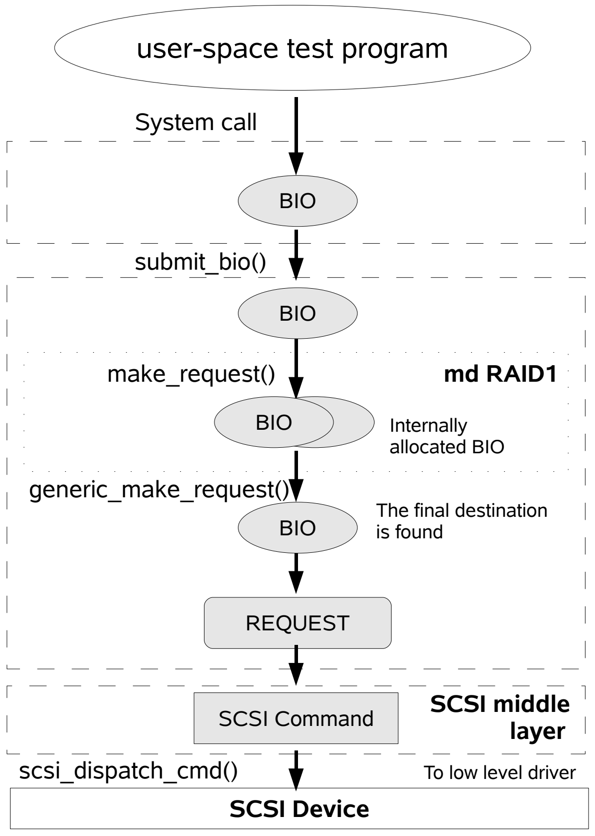

Figure 1: I/O flow from userspace to SCSI device

The software RAID driver resides in the middle of the block I/O layer. If software RAID is used, a `bio` is also generated by the software RAID driver, and the physical I/O destination and access length are stored in the `bio` accordingly.

Then, the `bio` is converted to a `struct request`. At that time, the `bi_sector` and `bi_size` of `bio` are stored in `sector` and `nr_sector` of `request`. `request` also includes a `rq_disk` member which links to `struct gendisk` representing the associated disk. The `gendisk` includes major and minor number of the disk.

Next, the `request` is sent to the SCSI middle layer from the I/O scheduler in a form of `struct scsi_cmnd`, which represents a SCSI command and it is linked to an associated `request`.

The physical I/O destination can be retrieved from `scsi_cmnd` through associated `request`'s `sector` member. A physical I/O access length can be retrieved from `scsi_cmnd`'s `request_bufflen` member. Also, the target device of the SCSI command is found in the associated `request`'s `rq_disk` member.

The SCSI command is issued to SCSI devices through SCSI lower level drivers. The command result is sent to SCSI middle layer and handled accordingly. When the command completed, the result is sent back to the block I/O layer.

The target SCSI command corresponding to the I/O request needs to be identified to inject a SCSI fault triggered by the I/O request represented by a `bio`. The SCSI fault injector will find the `bio` which corresponds with the trigger I/O request from the test program, find the `bio` sent to the SCSI middle layer, and finally find the SCSI command which corresponds to the `bio` requested from the block I/O layer. The target SCSI command is found by tracking the `bio` in the I/O flow described above, and comparing its members with a `scsi_cmnd` (see Figure 2).

Linux block I/O is classified into two types; cached I/O and direct I/O. Both types use `submit_bio()` function and `struct bio` to send a request to the block layer. The inode number of the file corresponding to the I/O request can be identified from `struct bio` for cached I/O or from `struct dio` for direct I/O.

At `submit_bio()`, which is the entry of block layer, the target `struct bio` can be distinguished by comparing block number, access length, inode number, and access direction taken from `bio` or `dio`, with those given by a test program. By tracking the `bio` in the I/O flow, the `bio` which contains the physical I/O destination, can be identified.

Before issuing a SCSI command to SCSI devices, the SCSI fault injector identifies the target SCSI command by comparing information taken from `scsi_cmnd` with that information given by the test program and taken from target `bio` found in the previous step. The compared information includes physical I/O destination, access length, access direction, and device major/minor number.

The `struct scsi_cmnd` representing the target SCSI command identified in this process is saved in a SystemTap variable for later use.

3.2 Injecting SCSI fault

First, the method to inject a fault for a single disk access is explained. Next, we show how each fault pattern de-

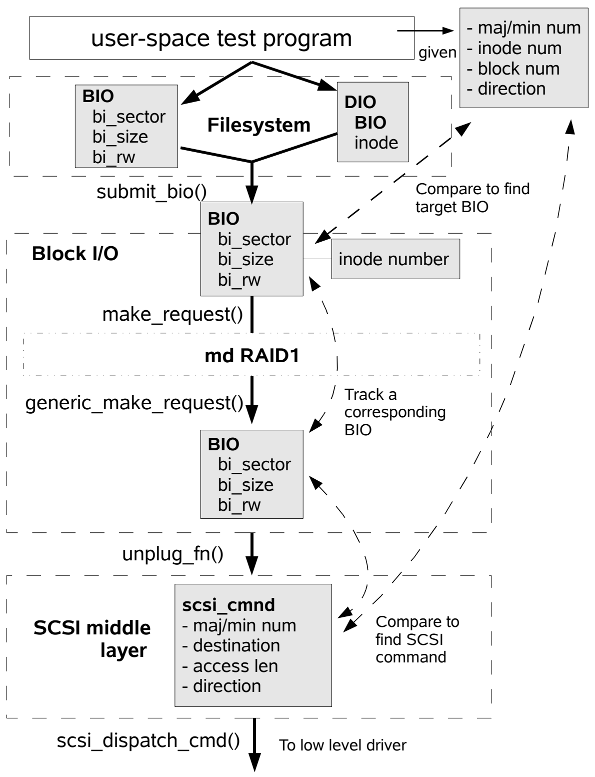

Figure 2: Block I/O tracking from BIO to SCSI command

scribed in Section 2.1 can be generated by changing the behavior in sequence.

3.2.1 Fault Injection Method

Once the target SCSI command is found, the SCSI fault injector modifies the target SCSI command both before issuing it to the lower layer driver and after returning the result, to simulate a SCSI fault.

The implementation details of "The SCSI device respond with an error" pattern and "The device does not respond" pattern described in Section 2.1 are as follows.

Error Response Case

To simulate device error response, modification of the result of the target SCSI command is needed to fake an error response to the upper layer. Also, actual data transfer generated by the SCSI command should not complete because when a real SCSI fault occurs, the DMA buffer may contain incomplete data (see Figure 3).

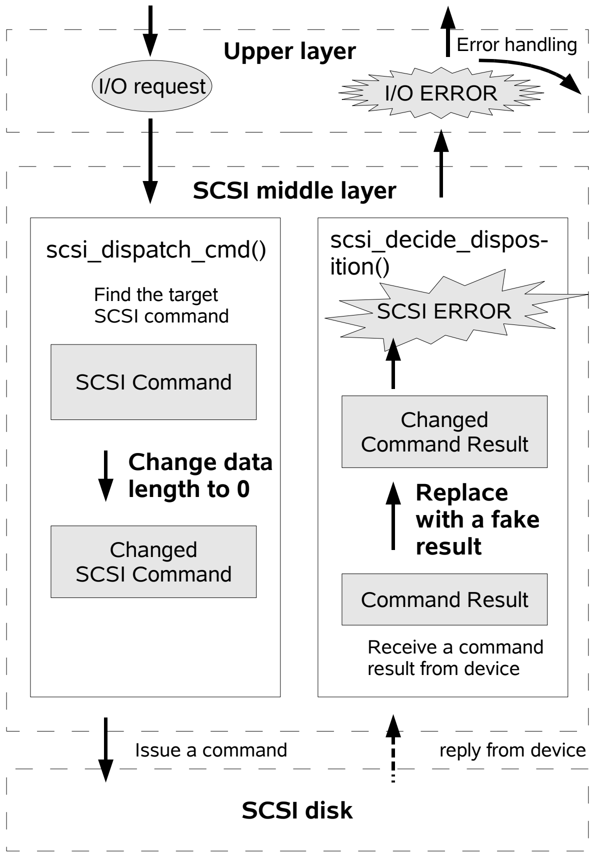

Figure 3: Simulating a fault with error response

For simulating incomplete data transfer to test whether the poisoned data is not sent to the upper layers, the data transfer length of the SCSI command is modified before issuing the SCSI command. When entering the `scsi_dispatch_cmd()` function before issuing a target SCSI command to lower layer driver, the data transfer length in `scsi_cmnd->cmnd` is overwritten to be zero. By this modification, the actual data transfer will not happen as expected.

To simulate error response, the result of target SCSI command needs to be modified before it is sent back to the upper layer. The SCSI command result is analyzed to be sent back to the upper layer in the `scsi_decide_disposition()` function. At the beginning of the function, to identify the target SCSI command, a `scsi_cmnd` which is given as an argument of the function is compared with the `scsi_cmnd` previously saved in a SystemTap variable. If it is the target command, the result stored in `scsi_cmnd` is modified using SystemTap so that the OS detects a medium error which is the most common HDD

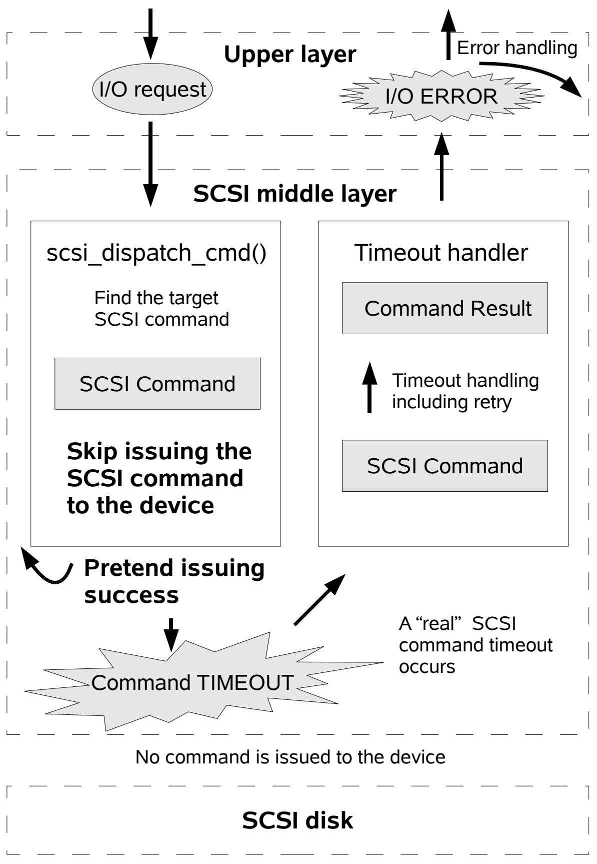

Figure 4: Simulating a fault with no response

error. More precisely, the following error values are stored in `scsi_cmnd` respectively; `(scsi_cmnd.result, scsi_cmnd.sense_buffer[2], scsi_cmnd.sense_buffer[12], scsi_cmnd.sense_buffer[13]) = (2, 3, 11, 4)`. This means "medium error, unrecovered read error, auto reallocated fail" which is one of the medium errors. Then, the changed status will be sent back to the upper layer.

No Response (Timeout) Case

The target SCSI command issued to the low level driver is skipped to simulate no device response (see Figure 4).

All SCSI commands are sent to low level drivers by the `queuecommand` operation in the Linux SCSI middle layer. If the `queuecommand` operation is skipped, the upper layer thinks that SCSI command is issued successfully. But actually it is not issued, consequently the SCSI command results in a timeout.

When the target `scsi_cmnd` is given as an argument of the `scsi_dispatch_cmd()` function, the `queuecommand` operation is skipped by using SystemTap.

3.2.2 Fault Patterns

A single error caused by a SCSI fault (medium error or timeout) can be injected as previously explained. The target SCSI command may be detected several times at `scsi_dispatch_cmd()` after accessing the faulty disk from the test program once, because the error handling code of the upper layer may retry the failed I/O request by an error handling code. The fault patterns described in Section 2.1 can be created by changing the SCSI command manipulation behavior at `scsi_dispatch_cmd()` in sequence as follows.

1. *Temporary read error*

2. *Temporary write error* – When a target SCSI command is detected at `scsi_dispatch_cmd()`, inject a fault just once. If the target SCSI command is detected at `scsi_dispatch_cmd()` again, the fault will not be injected any more.

3. *Read error correctable by write* – In this case, a fault is injected for read access to the target sector until error handling code tries to write data to the sector by tracking `scsi_dispatch_cmd()`. After the error handling code writes to the target sector, no fault will be injected because the error sector is assumed to be corrected.

4. *Permanent read error*

5. *Permanent read/write error* – When a target SCSI command is detected at `scsi_dispatch_cmd()` the fault is injected every time.

6. *Temporary no response on read access*

7. *Temporary no response on write access* – If a target SCSI command is detected at `scsi_dispatch_cmd()`, inject a fault by using "No Response (Timeout) Case" method. If the target SCSI command is detected at `scsi_dispatch_cmd()` again, the fault will not be injected again.

8. *Permanent no response on both read and write access* – A “No Response (Timeout) Case” fault will be injected every time a target SCSI command is detected at `scsi_dispatch_cmd()`.

Thus, all HDD fault patterns can be simulated.

4 Linux Software RAID Evaluation Using SCSI Fault Injection Test Tool

This section describes an example of evaluation using the proposed test tool by applying it to test error handling in the software RAID drivers.

The software RAID drivers were evaluated by injecting various SCSI faults to various RAID drivers and checking if SCSI middle layer and software RAID driver error handling code work properly.

First, the expected behavior of the fault handling code is explained for each fault pattern described in Section 2.1. Next, the test environment and procedure are explained. Finally, the test results and detected bugs are shown.

4.1 Expected Error Handling of Software RAID

The following are the expected error handling behavior of normal RAID array for each of the HDD fault patterns defined in Section 2.1. The test program will check if the system will behave this way when the fault is injected.

1. *Temporary read error* – The SCSI layer detects a read error and the error handler in the RAID driver retries the failed sector. The I/O completes successfully.

2. *Temporary write error* – The SCSI layer detects a write error and the RAID driver’s error handler will detach the failed disk immediately. The I/O completes successfully because the write access is issued to both failed disk and redundant disk. The error is recorded in syslog.

3. *Read error correctable by write* – After detecting a read error, the RAID driver retries once, which also fails, the RAID driver may try to write data to the failed sector and re-read from the failed sector. The failed sector will be corrected and the subsequent reads will succeed. The write fix behavior is recorded in syslog.

4. *Permanent read error* – This is the case that a read access fails even after sector correction attempts. As a result, the RAID driver will detach the failed device, read access is issued to another mirror disk, and the I/O completes successfully. The error is recorded in syslog.

5. *Permanent read /write error* – When a fault is triggered by read access, the error handling behavior is same as “Permanent read error.” When a fault is caused by write access, it is the same as “Temporary write error.”

6. *Temporary no response on a read access* – The SCSI layer detects timeout on target read access and the error handler of the SCSI layer retries the SCSI command. After the SCSI layer gives up, the read error is sent to the RAID driver. After that, the behavior is the same as “Temporary read error” and the I/O completes successfully.

7. *Temporary no response on a write access* – In this case, timeout detection and error handling by SCSI layer is same as “Temporary no response on a read access.” After the write access error is sent to the RAID driver, the behavior is the same as “Temporary write error” and the failed disk is detached and the I/O completes successfully.

8. *Permanent no response on both read and write access* – This case is the same as “Temporary no response on a read access” for timeout detection and error handling by SCSI layer, and the read/write request error is sent to the RAID driver. After that the behavior is the same as “Permanent read/write error” and the failed disk is detached and the I/O completes successfully.

4.2 Test Environment

The following test environment was used for the evaluation.

- A server with a single Xeon CPU, 4GB of RAM, and six 36GB SCSI HDDs.
- Fedora 7(i386) running Linux kernel 2.6.22.6
- The tested software RAID drivers were md RAID1, md RAID10, md RAID5, md RAID6, and dm-mirror in the following array conditions.

The hardware fault can occur on any of the disks constituting a software RAID volume. Evaluation was done for each case where the fault was injected when accessing each of the following disks in the software RAID volume.

1. *Active disk of redundant (normally working) array with spare disks* – In this case the failure occurs in one of the disks constructing a RAID array, which has redundancy with spare disks. If a disk is detached from this array as a result of the fault, the RAID array will start recovery using a spare disk.
2. *Active disk of redundant array without spare disks* – In this case, failure occurs in one of the disks constructing a RAID array, which has redundancy, but no spare disk. If a disk is detached from this RAID array as a result of the fault, it will lose its redundancy and become a degraded array.
3. *Active disk of degraded array* – In this case, failure occurs in one of the disks constructing a RAID array, which has no redundancy. If a disk is detached from this array as a result of the fault, the RAID array will collapse because this array has no redundancy.
4. *Active disk of recovering array* – In this case, failure occurs in one of the disks constructing a RAID array, on which the recovery process of the degraded array is running.
5. *Resyncing disk of recovering array* – In this case, failure occurs in a disk which is currently resyncing in the recovery process of the degraded array.

4.3 Test Procedure

The following procedure was performed in the evaluation.

- Install the OS on one of the SCSI disks. Using the rest of the SCSI disks, each of which has a single 8GB ext3 partition, construct a software RAID array.
- Inject various patterns of HDD fault defined in Section 2.1 to various conditions of software RAID array defined in Section 4.2. The fault injections are triggered by accessing a file located in the tested RAID device.
- Check if the target I/O request results in a SCSI error and inspect if the SCSI error is treated properly by error handler of the SCSI layer and the software RAID driver.

Since a set of operations needs to be repeated for all of the many test patterns, the evaluation used the test program to automatically perform the following works.

- Configure one of the five software RAID types (md RAID1, md RAID10, md RAID5, md RAID6, and dm mirror.)
- Set one of the five status of a RAID condition described in Section 4.2.
- Invoke one of eight SystemTap scripts corresponding to one of the HDD fault patterns defined in Section 2.1.
- Generate a SCSI I/O to inject a fault and log the results.
- Loop through all combinations to cover all patterns automatically.

The test program was implemented as a set of shell scripts.

4.4 Bugs Detected in the Software RAID Drivers Evaluation

All combinations of RAID volume types, fault patterns, and RAID volume conditions were tested. The evaluation revealed the following bugs related to error handling of software RAID.

md RAID1 issue is as follows:

- The kernel thread of md RAID1 could cause a deadlock when the error handler of md RAID1 contends with the write access to the md RAID1 array [2].

md RAID10 issues are as follows:

- The kernel thread of md RAID10 could cause a deadlock when the error handler of md RAID10 contends with the write access to the md RAID10 array [2].

- When a SCSI command timeout occurs during RAID10 recovery, the kernel threads of md RAID10 could cause a md RAID10 array deadlock [3].
- When a SCSI error results in disabling a disk during RAID10 recovery, the resync threads of md RAID10 could stall [4].

dm-mirror issue is as follows:

- dm-mirror's redundancy doesn't work. A read error detected on a disk constructing the array will be directly passed to the upper layer, without reading from the other mirror. It turns out that this was a known issue, but the patch was not merged [1].

All these bugs have already been reported to the community and a fix will be incorporated into future kernels. Many bugs found in our evaluation were caused by race conditions between the normal I/O operation and threads in the RAID driver. Probably such bugs were hard to detect. However the proposed test tool using SCSI fault injection was able to find such issues.

5 Conclusion and Future Works

The evaluation result proves that the proposed test tool, which is a combination of the SCSI fault injector and test programs, has the powerful functionality to inject various patterns of HDD fault on various configurations of a software RAID volume to be used for error handler testing. Especially, the flexible user interface of the proposed SCSI fault injector, which existing test methods do not have, realizes a user-controllable fault injection. Also, by applying the proposed test tool, some delicate timing issues in Linux software RAID drivers are found, which are difficult to detect without thorough testing. Therefore, it can be concluded that the proposed test tool using SCSI fault injection is useful for systematic SCSI block I/O test.

The authors are planning to propose the SCSI fault injector to SystemTap community so that the injector becomes available as a SystemTap-embedded tool. Contribution of the injectors for other drivers are welcome as the wider set of fault injectors can form a more generalized block I/O test framework.

Acknowledgments

This project is supported by the Information Technology Promotion Agency (IPA), Japan.

References

[1] Announcement of SCSI fault injection test framework (mail archive). `http://marc.info/?l=linux-raid&m=120036612032066&w=2`.

[2] Bug report of md RAID1 deadlock problem (mail archive). `http://marc.info/?l=linux-raid&m=120036652032432&w=2`.

[3] Bug report of md RAID10 kernel thread deadlock (mail archive). `http://marc.info/?l=linux-raid&m=120289135430654&w=2`.

[4] Bug report of md RAID10 resync thread deadlock (mail archive). `http://marc.info/?l=linux-raid&m=120416727002584&w=2`.

[5] Fault Injection Test project site on SourceForge. `https://sourceforge.net/projects/scsifaultinjtst/`.

[6] `scsi_debug` adapter driver. `http://sg.torque.net/sg/sdebug26.html`.

[7] SystemTap project site. `http://sourceware.org/systemtap/index.html`.