IRON FOR JFFS2

Raja Ram Yadhav Ramakrishnan, Abhinav Kumar

{ rramakrishn2, kumar8}@wisc.edu

ABSTRACT

Flash memory is an increasingly common storage medium in embedded devices, because it provides solid state storage with high reliability and high density, at a relatively low cost. There are only very few file systems that are tailored to work on the Flash memory, JFFS2[1] and YAFFS2, the most popular among them. Our work aims at studying the behavior of JFFS2 [1], in particular understanding the policies adopted by it in case of the malfunctioning of the Flash device. We introduced several failures at the device driver level and studied how JFFS2 [1] reacts to these failures and have recorded our observations.

INTRODUCTION

Flash memory is being increasingly used in several applications such as the internal memory of smartphones. Among the various flash file systems YAFFS(Yet Another Flash File System) and JFFS2 [1](Journaling Flash File system) are currently dominant. Between the two, there aren't many differences but JFFS2 [1] has some unique features like on-the-fly compression and decompression, effective use of the Out-of-Band(OOB) etc. Hence if the storage capacity is less JFFS2[1] is preferred. The properties of Flash devices are quite different from that of Char or Block devices. For Eg. Block devices consist of sectors whereas Flash devices consist of Eraseblocks, Block devices do not have an erase operation and most importantly sectors of Block Devices are devoid of a wear-out property. But in Flash devices, eraseblocks wear out and become unusable after a certain number of erase cycles. Flash File Systems must possess wear leveling, which is a property which distributes the writes throughout the device such that the number of re-writes and erases are bounded. Thus the characteristics and policies adopted by a Flash File System are different as compared to traditional file systems. Very little is known about these policies like how they react to read or write failures, bit corruptions etc.

Our aim in this project was to perform several tests on the File system to understand its policies during failures and to our knowledge such an analysis has not been done for Flash File systems. We picked JFFS2[1] because of its unique properties listed earlier. Such an analysis has several advantages. We can get a good idea about the File System's policy without actually looking at the implementation. The approach of using a pseudo-driver enables us to perform these tests on other Flash File Systems too without almost any change in the pseudo-driver code. Upon performing these tests, we were able to find out the behavior of JFFS2 during several fault scenarios and also able to make certain observations which were previously unknown about JFFS2.

In the following sections, we describe the related work to this project, our implementation in much more detail and our results.

RELATED WORK

This work is similar to the IRON file systems paper by Vijayan Prabhakaran et al.[2] IRON paper focuses primarily on the study of the modern disk failures and the way several file systems handle their failure. Our work is much similar to their work in that we study the JFFS2 file system for the various failures that are possible in Flash devices. Also, the file systems that were tested were mostly of Journaling nature whereas JFFS2 is a log structured file system. Our work is the first in performing such a study focusing on the Flash devices and studying the Flash file system for the various failure cases.

BACKGROUND

In NAND storage, memory is arranged as an array of pages. A memory page consists of 256/512 bytes of data and 8/16 bytes of out-of-band area (spare area). The spare area is used mainly for storing the checksum and file system dependent data. For example, JFFS2 uses the spare area to store the checksum and also its cleanmarkers. Cleanmarkers occupy 8 bytes in the first page of an erased block. If an erase operation was successful, then these 8 bytes will be set to pre-defined values. Hence a good Flash File System also makes efficient use of this out-of-band area.

Flash Devices are quite different compared to block or char devices due to several reasons listed in this report earlier. But most importantly in flash Devices erase blocks wear out and become unusable after a certain number of erase cycles. Flash File Systems must possess wear leveling. Hence the most common approach adopted in devices such as USB Flash Drives, SD Cards is to use a File Translation Layer (FTL) which emulates a block device on top of NAND storage and then use one of the traditional file systems such as ext2 etc. But such an approach (as shown in fig 1) is very **inefficient**. This is because a file translation layer acts as a journaling file system. Then, making use of another journaling file system on top of this FTL is redundant. Instead JFFS2 which is a log structured file system is used directly on top of the NAND chip drivers. Since it is log-structured it provides wear leveling too.

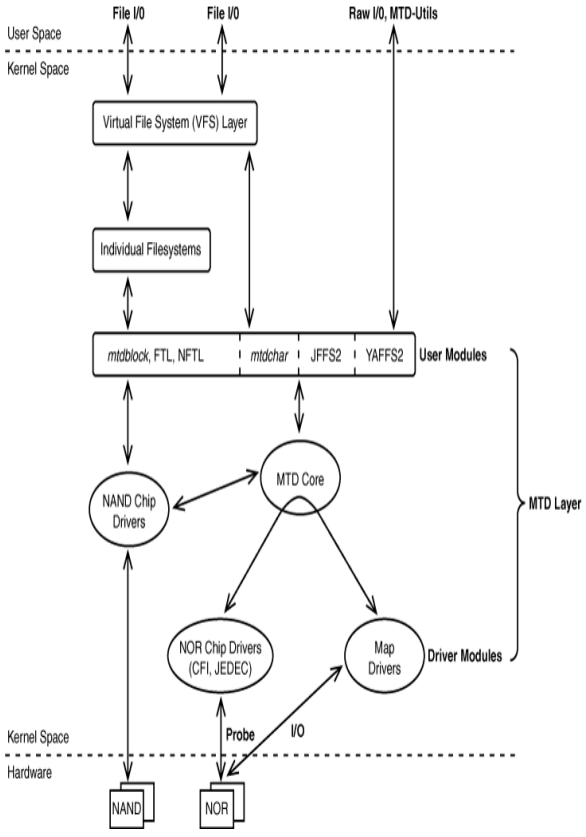

Fig 1: Flash File system software stack

IMPLEMENTATION

We used a simulator called nandsim which comes as a part of Linux kernel. NAND simulator (nandsim) is a powerful debugging and development tool which simulates NAND flashes in RAM or a file. nandsim can simulate various errors and report wear statistics, which is extremely useful when testing how flash software handles errors. Using nandsim, we created a pseudo device which works similar to a raw NAND Flash device. From a file system point of view, it appears to JFFS2 as if it was writing to an actual Flash device. We then modified the code for nandsim to introduce errors to the File System above and studied how JFFS2 reacts to the various common error scenarios that are possible in Flash devices.

The exact steps of implementation are as follows. We created a pseudo-device using nandsim and created a file. We then wrote a long pattern of data in the file whose probability of occurrence in metadata is very less. We did this to identify a particular data block in which we want to introduce the various kinds of failures. In the device-driver code of this pseudo device, during read, write or erase of the block, we searched for the occurrence of this particular pattern and once we found that pattern we introduced the errors. To make sure the data is read from the device and written to the device as opposed to just the buffer, we unmounted the pseudo device and mounted the device again which ensures that fysnc() function call is made and the data was flushed to the flash device.

The technique of pattern matching was simple enough for identifying a particular page to fail. We did not opt to fail a random page because we did not want to introduce a failure at the time when the device is mounted. We wanted to ensure that the device functions normally in all the cases except for a particular page which contains this pattern.

The following were the list of cases that we tested

1. Read failure of an page
2. Single bit flip during read of a data page
3. Multiple bit flips during read of a data
4. Single bit flip in the OOB area
5. Multiple bit flips in the OOB area
6. Write failure of a page

These were the tests that we performed to test the most common case of reads and writes. We were also interested in understanding the policies that were adopted during the Garbage collection path to handle failures. So we tried to fail the reads, writes and erasures along the Garbage collection path and have recorded our observations. The following were the tests that we performed along the GC path.

1. Read failure along GC path
2. Write failure of a page along GC path
3. Erase failure of a page along GC path

We will briefly explain the tests that we performed and list a set of observations

Read failure of an entire page:

This test was performed as its quite common for a read() call to fail. The reason could be corruption of data, wearing out of the erase block etc. We created a file in the pseudo device, wrote a pattern into the file and saved it. In the device-driver code of nandsim, during the read of data page we returned an error code simulating the case when a particular page of a flash device has worn out and cannot be read. When we issued a read() on the file, we

received the same error message that we introduced implying the case that JFFS2 was able to detect that the read failed and as a recovery mechanism it propagated the error to the layer above. But there was no retry mechanism observed. Fig 2 shows the error message that we observed in the kernel log

```
JFFS2error:(10411)
jffs2_get_inode_nodes: cannot read
512 bytes from 0x01ff8200, error
code: -5.

JFFS2error:(10411)jffs2_do_read_inode
_internal: cannot read nodes for ino
2, returned error is -5

Returned error for crccheck of ino
#2. Expect badness...
```

Fig 2: Error returned for read failure

Hence the following are our observations from this test case

1. JFFS2 was able to detect page failure
2. Error propagation was observed
3. No retry was attempted

Single bit flip in a data page:

This is another error which is commonly observed in all storage media. A single bit might get flipped or corrupted and as a result it leads to a read failure. In this case when the pattern was observed during the read of a page, we flipped a single bit of the page and returned the data to the JFFS2 layer. We observed that JFFS2 code was able to identify the single bit flip and was able to correct that particular bit change using the checksum that it stored in the OOB area of that page. The data that JFFS2 returned was the same as what was present in the file. No error message was observed in the kernel log.

Hence the following are the observations for this error case

1. Error was detected
2. Recovery using checksum

Multiple bit flip in a data page:

We wanted to extend the previous test case to fail multiple bits. Once again this is also a common occurrence in storage media and we wanted to observe JFFS2's behavior under this failure. Similar to the case above, we flipped multiple bits and returned the data to JFFS2. In this case, we observed a kernel message as shown below in Fig 3

```
JFFS2 notice: (8889) read_dnode:
wrong data CRC in data node at
0x01ff8200: read 0x1aaee62e,
calculated 0x6cae3da6
```

Fig 3: Error returned for multiple bit flip

This error message was observed when we tried to remount the device. As the error message shows, JFFS2 calculated the checksum for the data in the page and compared it with the checksum that it stored in OOB area. But JFFS2 did not attempt any sort of recovery from this error. This error was not propagated to the levels above. There was no retry that was attempted. The mount succeeded and the read of the file did not show any error but the file was shown to be empty. Looking through the code, we observed that since JFFS2 cannot correct this through checksum, it assumed that the data page has worn out and marked the page obsolete. But we feel that the user should have been given some sort of notification. The code snippet is given below in Fig 4 for reference

```
static inline int read_dnode(struct jffs2_sb_info *c, struct jffs2_raw_node_ref *ref,

                         struct jffs2_raw_inode *rd, int rdlen,

                         struct jffs2_readinode_info *rii)

{

    int ret = 0;

    ********

    ********

    if (len >= csize && unlikely(tn->partial_crc != je32_to_cpu(rd->data_crc)))

    {

            JFFS2_NOTICE("wrong data CRC in data node at 0x%08x: read %#08x,
    calculated %#08x.\n",ref_offset(ref), tn->partial_crc,
    je32_to_cpu(rd->data_crc));

            jffs2_mark_node_obsolete(c, ref);

            goto free_out;

    }

    **********

    **********

    free_out:

            jffs2_free_tmp_dnode_info(tn);

            return ret;

}
```

Fig 4: Code snippet for behavior during multiple bit flips

The following are the observations from this interesting case

1. Error was detected using checksum
2. No error propagation
3. No retry

Bit flips in OOB area:

We also tried to flip the bits in the OOB area. The observations were similar to the case of the bit flips in the data page.

Write failure: To simulate the case when the write to a page in the device fails due to the wearing out of the device, in the device driver coded uring the flushing of data to the device, we returned an error code to the JFFS2 layer. As JFFS2 is a log-structured file system, as soon as it writes the data to the buffer, success message was sent back to the write() system call. But during unmounting (to be precise during the fsync() call of unmounting), JFFS2 tried to flush the buffer contents to the pseudo device where we injected an error code. When the flush to device failed, JFFS2 retried the write to that page infinitely in a while loop till it succeeded. But since our driver returned an error code whenever the pattern was observed, JFFS2 code was stuck

in an infinite while loop and led to the crash of the kernel. We feel that a kernel crash due to a write failure in a page is not appropriate. When we modified our code to fail the block a finite number of times, it attempted the same finite number of times and then finally succeeded in writing the data to the disk. The following are the observations from this case

1. Error detection
2. Error propagation was observed
3. Infinite retry

Fig5 shows the code snippet where infinite retry is done

```
int jffs2_flush_wbuf_gc(struct jffs2_sb_info *c, uint32_t ino)
{
******
while (old_wbuf_len && old_wbuf_ofs == c->wbuf_ofs)
        {
                mutex_unlock(&c->alloc_sem);
                D1(printk(KERN_DEBUG "jffs2_flush_wbuf_gc() calls gc pass\n"));
                ret = jffs2_garbage_collect_pass(c);
                if (ret) {
                        /* GC failed. Flush it with padding instead */
                        mutex_lock(&c->alloc_sem);
                        down_write(&c->wbuf_sem);
                        ret = __jffs2_flush_wbuf(c, PAD_ACCOUNTING);
                        /* retry flushing wbuf in case jffs2_wbuf_recover
                           left some data in the wbuf */
                        if (ret)
                                ret = __jffs2_flush_wbuf(c, PAD_ACCOUNTING);
                        up_write(&c->wbuf_sem);
                        break;
                }
*****
```

Fig 5: Code snippet for infinite retry during write failure

Tests along GC Path:

The garbage collection mechanism is an integral part of any log structured file system. It is even more vital in the case of flash file systems because of the limited amount of space available. Hence we performed several tests along the GC path to find out how JFFS2 reacts to failures. The following were the set of tests that we performed along the GC path and the corresponding observations were recorded.

Read failure along GC path:

We were not able to simulate the case of a read failure along the GC path as the simulator was always reading the data from the buffer instead of reading from the device.

Write failure along GC path:

This was one of the most difficult cases to simulate as there was no definite pattern while writing during garbage collection. The live blocks are just copied from one region of the storage into another. We adopted a slightly different approach for this test. We searched for the file name in the metadata which is written and we count this occurrence. The first occurrence is the creation of the file but the second occurrence must be that because of garbage collection. Hence we failed write when this 'file name' pattern was found for the second time. The results of this test case were very similar to what happened during the write failure of a data page. JFFS2 tried to write infinitely and the kernel crashed resulting in a reboot.

Erase failure along GC path:

The erase operation is a critical operation for Flash Devices. The erase blocks are periodically erased during garbage collection and hence failing the erase operation is an interesting case. Our analysis would have been incomplete without simulating a failure along this path. During garbage collection, when the pages were erased by marking the data as 'FF' and setting the clean marker, we introduced failure. We failed all the pages during this erase operation as identifying a particular block to fail was difficult. In this case, the garbage collector tried to erase all the blocks and failed to mount the device and gave out an error indicating that there is no space left on the device as shown in Fig 6.

```
Erase at 0x01ffc000 failed immediately:
errno -1
No space left on device
```

Fig 6: Error returned during erase failure

failures by introducing a bound on the number of times we fail the erase operation.
It was observed that under this failure case the mount was successful.

RESULTS:

We now summarize the results of all the tests that we performed on JFFS2 in fig 7 below. These results helped us to conclude quite a few things about JFFS2 which we elaborate in the next section.

Test case	Error Detected?	Error correction	Observations
Page read failure	Yes	Appropriate	1. Error was detected 2. Error was propagated 3. No retry
Single bit flip	Yes	Appropriate	1. Error detected using checksum 2. Error corrected using checksum
Multiple bit flips	Yes	Inappropriate	1. Error detected using checksum 2. No error propagation 3. No retry
Write failure	Yes	Inappropriate	1. Error was detected 2. Infinite retry leading to kernel crash 3. Error was propagated
Erase failure	Yes	Appropriate	1. Error was detected 2. Error was propagated 3. No retry was attempted
Write failure along GC path	Yes	Inappropriate	1. Error was detected 2. Infinite retry leading to kernel crash 3. Error was propagated

Fig 7: Table summarizing the results of various tests

CONCLUSION:

JFFS2 has several attractive properties like on-the-fly compression and decompression, efficient usage of the out-of-band area and a good garbage collection mechanism making it a good choice for NAND storages where the storage capacity is limited. From our **test-driven** study of the JFFS2 flash file system, we observed that the techniques used by JFFS2 to detect error are checksum and the error returned by the lower level driver. The major recovery techniques are error propagation, retry and checksum. Though the recovery mechanisms were appropriate in most of the cases, we observe that in few cases like multiple bit failure and write failure, the techniques were inappropriate. By working on this project, we gained a good understanding about the File System architecture and some internals about JFFS2 including details like how it is tailor made for working on NAND flash devices.

FUTURE WORK:

Though we performed a very detailed study of JFFS2, we would like to spend more time studying and conducting some more tests along the GC path. Also, we were not able to fail specific metadata blocks such as inode, i-node map etc because of JFFS2's on-the-fly compression.

This would enhance our knowledge about JFFS2.

REFERENCES:

[1] David Woodhouse, Red Hat Inc**.** **JFFS : The Journalling Flash File System**

[2] Vijayan Prabhakaran, Nitin Agrawal, Lakshmi Bairavasundaram, Haryadi Gunawi, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. **IRON File Systems**

[3] Sreekrishnan Venkateswaran**. Essential Linux Device Drivers**

[4] Jonathan Corbet, Alessandro Rubini,Greg_Kroah-Hartman. **Linux Device Drivers**, Third Edition

[5] http://www.linux-mtd.infradead.org/