Write Back Routine for JFFS2 Efficient I/O

Seung-Ho Lim[1], Sung-Hoon Baek[1], Joo-Young Hwang[2] and Kyu-Ho Park[1]

[1] Computer Engineering Research Laboratory,
Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology
{shlim,shbaek}@core.kaist.ac.kr, kpark@ee.kaist.ac.kr,
[2] Embedded OS Lab.
Samsung Electronics
jooyoung.hwang@samsung.com

Abstract. When flash memory is used as a storage in embedded systems, block level translation layer is required between conventional filesystem and flash memory chips due to its physical characteristics. A far more efficient use of it is the design of a filesystem itself without no extra layer of translation. However, since flash filesystem does not use block device layer, it cannot utilize deferred I/O although deferred I/O enhances write latency by delaying the flushing jobs. Linux operating system generally uses the write back routine for deferred I/O using kernel thread, which writes back dirty pages and buffers through the block device layer. In this paper, we design and implement efficient I/O for JFFS2 flash filesystem based on flash memory. For this, we first analyze the write procedure of JFFS2 filesystem in detail, and derive the drawback and overhead. Then, we design the flash write back routine for deferred I/O. We apply it to the Linux JFFS2 by implementing $ffflush$ and $flash\_writeback$ kernel thread. The designed flash write back routine can reduce average write latency when the kernel buffers are enough to get the users data.

1 Introduction

Flash memory has become an increasingly important component as a nonvolatile storage media because of its small size, shock resistance, and low power consumption[1]. In nonvolatile memories, NOR flash memory provides fast random access speed, but high cost and low density, compared with NAND flash memory. NAND Flash has advantages in large storage capacity and relatively high performance for large read/write in contrast to NOR flash. Recently, the capacity of NAND flash memory for one chip becomes 2GB, and this will be increased quickly. Based on the NAND flash chip, solid state disk was developed and this could be used as a storage system in labtop computer[3]. Therefore, NAND flash is widely used as data storage in embedded systems and also likely to be used for PC based systems in the near future. Specifically, NAND Flash memory chips are arranged into blocks, and each block has a fixed number of pages, which is the unit of read or write. A page is further divided into a data region for storing data and a spare region for storing the status of the data region. In first

generation, the typical page size was 512 bytes, the additional spare region was 16 bytes, and the block size was 16 KB, which was composed of 32 pages. As its capacity grew, the page size of the next generation became 2 KB with an additional 64 bytes spare region, and the block size became 128 KB.

Due to the flash memory characteristics, form of Electrically Erasable Read Only Memory (EEPROM), no in-place update is allowed. This means that when data is modified, new data must be written to an available free page in another position, and this page is considered as a live page. The page which contains old data is considered as a dead page. As time passes, a large portion of flash memory is composed of dead pages, and the system should reclaim the available free pages for write operations. The erase operation makes free pages available. However, because the unit of an erase operation is a block, which is much larger than a write unit, this mismatch causes an additional copy operation of live pages in erasing the block somewhere else. This process is called garbage collection. To address these problems, a flash translation layer has been introduced between the conventional filesystem and flash memory[5]. This block level layer redirects the location of updated data from one page to another page and manages current physical location of each data in the mapping table. The mapping between the logical location and physical location can be maintained either at the page level (FTL)[2] or at the block level (NFTL)[6]. However, the use of an conventional filesystem has many performance restrictions when using on the flash memory because conventional filesystems are designed for disk based storage system.

A far more efficient use of flash memory as storage would be possible through the use of a filesystem designed specifically for use on such devices, with no extra layer of translation in between. One such design is Journaling Flash File System 2[9]. The JFFS2 is a log-structured filesystem which stores nodes containing data and metadata to every free region in flash chip, sequentially. This sequential write is performed for each flash block. One of the most serious problem in JFFS2 is very poor read/write response time at users feeling. When writing new user's data, JFFS2 makes new node which contains both inode information and data, writes to the flash medium using flash driver interface directly. Then, node tree structure is managed and maintained in the main memory using the specific data structures such as jffs2 inode cache and node fragment lists. Whenever the write is performed from users, it is not returned from kernel until all the data are written out to flash memory. There are two reasons that JFFS2 uses synchronous I/O scheme for write operation. First, JFFS2 does not use block device layer, so it cannot use the buffer cache mechanism. Although JFFS2 maintains the compatible interface with the virtual file system(VFS) layer and uses the page cache from the VFS, the deferred I/O mechanism in Linux operating system should use both page cache and buffer cache. Second, flash memory should preserve the write sequentiality for each flash block, which is from the inherent characteristics of flash memory. JFFS2 should preserve the write sequentiality of the node to be written. This write transaction mechanism which is implemented in JFFS2 gives much latency and response time to users and degrades the overall system read/write performance.

In this paper, we design and implement the write back routine for JFFS2 filesystem on the flash memory storage. Linux OS generally use the write back routine for deferred I/O using *pdflush* and *kupdate* kernel thread, which writes back the dirty memory pages and buffers to the real storage medium through the block device layer. For this, we first analyze the write procedure of the JFFS2 filesystem in detail, and derive the drawback and overhead. Then, we design the flash write back routine for the flash filesystem deferred I/O whose method is similar to the Linux *pdflush* and *kupdate* operation for dirty buffers. We implement the flash flush operation and apply to the JFFS2 flash filesystem. The remainder of the paper is organized as follows: Section 2 describes the background of Linux deferred I/O mechanism. Section 3 explain the designed write back routine for JFFS2, and Section 4 describe the performance evaluation of write back routine. Finally, we conclude in Section 5.

2 Background

In this section, we describe the write procedure in Linux kernel internals[11]. The overall procedure is described in Figure 1. The *write*() system call involves moving data from the user mode address space of the calling process into the kernel data structures, and then to storage. There are two important data structures in implementing filesystem, the one is file_operations and the other is address_space_operations. The write method of the file_operations object permits each filesystem type to define a specialized write operation. It is the a procedure that basically identifies the blocks involved in the write operation, copies the data from user mode address space into some pages belonging to the page cache, and marks the buffers in those pages as dirty. Generally, filesystems implement the write method of the file object, which is inherited by the file_operations, by means of the *generic_file_write*() function. The prepare_write and commit_write methods of the address_space_operations object specialize the generic write operation implemented by *generic_file_write*() for files. Both of them are invoked once for every page of the file that is affected by the write operation. Each block based filesystem implements simply a wrapper for common function. For example, EXT2 filesystem[7] implements the prepare_write function by means of the following function:

```
int ext2_prepare_write(struct file *file,
                       struct page *page, unsigned from, unsigned to) {
    return block_prepare_write(page,from,to,ext2_get_block)
}
```

The *ext2_get_block*() function translates the block number relative to the file into a logical block number, which represents the position of the data on the physical block device. The *block_prepare_write*() function takes care of preparing the buffers and the buffer heads of the file's page. Once the prepare_write function returns, the *generic_file_write*() function updates the page with the data stored in the user mode address space using *copy_from_user*() macro. Next,

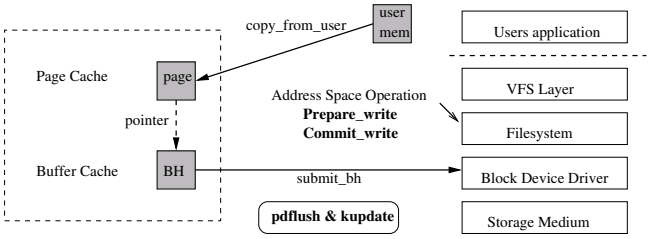

Fig. 1. Write Procedure and data structures in Linux Kernel

the *generic_file_write*() function invokes the commit_write function of the address_space_operations. The commit_write function can be also implemented by a wrapper for the common function, *block_commit_write*(). It considers all buffers in the page that are affected by the write operation; for each of them, it sets the BH_Uptodate and BH_Dirty flags and inserts the buffer head in the BUF_DIRTY list and in the list of dirty buffers of the inode. After the commit_write function returns, the *generic_file_write*() function does the same job until every page is updated by the prepare_write and commit_write methods.

The dirtied pages and buffers are not sent to the block layer and flushed to the storage medium immediately. Linux allow the deferred writes of dirty buffers into block devices, since this noticeably improves system performance. At the system's aspect, I/O bandwidth is increased and at the users aspect, the write response time is reduced. A dirty buffer might stay in main memory until the last possible moment. However, when the system failure occurs, the updated data are lost if the dirty buffers are not flushed to permanent storage. Also, the size of main memory would have to huge at least as big as the size of the accessed block devices. Therefore, the dirty buffers are flush to storage under the following conditions; The buffer cache gets too full and more buffers are need, too much time has elapsed since a buffer has stayed dirty, or *sync*() system call is invoked. Among these conditions, first two conditions are occurred by the system's request. To perform flushing the dirty buffers, Linux runs kernel thread called *pdflush* and *kupdate* in Linux kernel 2.6. While the *pdflush* kernel thread is activated when there are too many dirty buffers or when more buffers are needed, the *kupdate* kernel thread is introduced to flush the older dirty buffers. Therefore, the *kupdate* thread is a periodic timeout function that flush the dirty buffers to the real storage medium.

3 Write Back Routine for JFFS2

In this section, we explain the implementation of JFFS2 write back routine. Since the read/write operation of flash filesystem is strongly related to the filesystem architecture, our implementation of write back routine is only dedicated to JFFS2 filesystem. However, the design concept can be generalized to all of the flash memory based file system. Actually, it is our future work.

There are some implementing issues in applying the write back routine in flash filesystem. First, like the *pdflush* kernel thread and *kupdate* function, the kernel thread and writeback function are required whose roles are transferring the deferred data to the lower layer for real flushing, similar to previous ones. Second, another data structure is required that contains the information to be deferred data just like a buffer_head in general system, since JFFS2 cannot utilize buffer_head data structure. Lastly, the connecting code should be implemented between JFFS2 write code and writeback kernel thread code. The jffs2_prepare_write and jffs2_commit_write method of the JFFS2 address_space object do the jobs. The Figure 2 shows the overall implementation of JFFS2 write back routine.

```
struct jffs2_flash_bh \{
    struct list\_head queue\_list;
    unsigned short type;
    struct page *page;
    unsigned start;
    unsigned end;
    unsigned size;
}
```

The defined structure jffs2_flash_bh is shown in above. It should have variables that jffs2 prepare_write and commit_write function can deliver the important parameters. In the structure, queue_list represents the jffs2_flash_bh list that is to be scheduled for real flushing, and the type represents JFFS2 node type. The page, start, end and size means page pointer, page start address, page end address, and written size, respectively. These are the arguments from generic_file_write function. When $jffs2\_prepare\_write()$ function is called, the jffs2_flash_bh is allocated for node and the member variable, *page, in jffs_flash_bh is is set to the value from the $generic\_file\_write()$. However, there is no data write and only node contains metadata in the $jffs2\_prepare\_write()$, the variables, start and end, have null values. Jffs2_flash_bh is inserted into the flash_bh list for deferred write. The flash_bh list, in here, contains all the deferred jffs2_flash_bh structure objects. It is generated when write back kernel thread is initialized. which is done using the queue_list and Linux list_head macros. Then jffs2_prepare_write returns. The $jffs2\_commit\_write()$ does the similar job, which also allocates one jffs2_flash_bh, and so on. In addition, the start, end and size is set for the data writing, then the constructed structure is inserted into the flash_bh list. After that, the VFS related inode information is updated such as update time and inode size. Since original JFFS2 writes page data to flash memory immediately, pages are not set to anything in JFFS2 address_space operations. We should prolong the page lifetime until the data in pages are written to flash memory by the write back routine. It is done by setting pages appropriate flags and not releasing page memory. Then, $jffs2\_commit\_write()$ returns.

JFFS2 write back routine is composed of $fflush$ kernel thread, and a timer function called $flash\_writeback$. The $fflush$ kernel thread is created during system initialization. It executes $fflush()$ function, that gets the argument func-

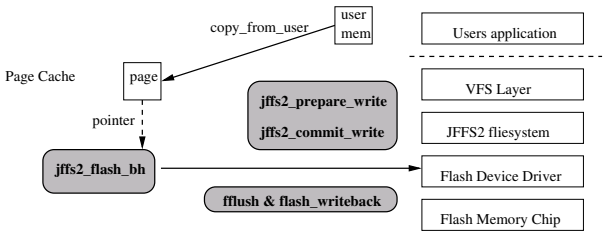

Fig. 2. Write Back Routine of JFFS2 in Flash Memory based Storage

tion, called $wb\_flash()$, which selects some nodes from flash_bh list and forces an update of corresponding flash memory pages and blocks. This argument function is transferred from the flash_writeback timer function. The flash_writeback timer function is created by the Linux timer interface, $mod\_timer()$, that sets the timer interval and the function to be executed when the timer is expired. When it is created, list_head for the list management of jffs2_flash_bh, called flash_bh, is generated. In $mod\_timer()$ setting, we set the $wb\_flash()$ to be executed, later it is executed when $fflush$ thread has scheduled. The $wb\_flash()$ function is key function that actually schedule the jffs2_flash_bh elements in list which are gatherd from $jffs2\_prepare\_write()$ and $jffs2\_commit\_write()$. In wb_flash, it checks whether the flash_bh contains any list element or not. If it is, it gets one element from the list and tries to write it to flash memory. It makes appropriate type of JFFS2 node, fills node variables with metadata and data, and sends it to lower layer by calling $jffs2\_write\_dnode()$ function. If the write is done with success, the element is eliminated from the list. The $wb\_flash()$ function iterates the same job until timer function is expired or threshold number of elements that is written in this time becomes over. The list scheduling and merging can be considered while dealing the elements. Each data size and offset in the element is likely to align with the Linux page size and offset since address_space operation in Linux is performed with page aligned. Therefore, the offset alignment of elements in the list is likely to be continuous order, which can be merged. This operation is done until the element cannot be merged into previous ones. Or, the merging should be stopped when merged data size is exceed the size of flash block because JFFS2 node should not be exceed the size of flash memory block. The JFFS2 node can be generated with the merged element, and flushed into flash memory. Since flash memory should ensure the sequential write order for each flash block, the merged node should be written immediately. Merging the elements reduce the number of JFFS2 nodes, thus it can reduce the system overhead not only managing the node fragments list but also flash memory storage itself.

Finally, special treatments are required for special filesystem operations. When $fsync()$ system call is called, the file should be flushed into flash memory immediately. In this case, jffs2_fsync method in file object tries to write jffs2_flash_bh elements which are related to the file by finding them in the flash_bh

Experimental Platform	NAND Flash Memory Characteristic.
Platform : OMAP 5912 OSK	Part Number : KFG1G16U2M
Memory : SDRAM 32MB	Block Size : (128K+4K)Bytes
CPU : ARM9 168MHz	Page Size : (2K+64)Bytes
Flash Memory : OneNAND 128MB	Page Read Time : 30us
OS : Linux 2.6.9	Page Program Time : 220us
MTD Snapshot Ver. : 2005/10/22	Block Erase Time : 2ms

Table 1. Experimantal Platform Environment and NAND Flash Memory Characteristics.

list, making JFFS2 nodes and programming to flash memory. When filesystem is unmounted, all the list elements should be written to flash memory.

4 Performance Evaluation

We have experimented JFFS2 write performance with our implementation of write back routine for JFFS2 deferred I/O. For the experiments, we use the embedded system which is composed of ARM CPU based system with ARM9 192MHz clock frequency and 32MB main memory[12]. The used NAND flash memory is newly designed memory chip, OneNAND[4]. OneNAND has NOR interface, however, the memory sell is composed of NAND array. It can be connected to the memory system bus, and used for storage. Therefore, there is no problem the performance test to see the flash memory based flash filesystem. The developed and experimented software platform is Linux 2.6.9. In the Linux kernel, we have modified and implemented the JFFS2 write back routine. The experimental environments are summarized in Table 1. The test program we use is the tiobench[13] benchmark program, which is designed to test I/O performance with multiple running threads. The tiobench generates multiple threads that do the read/write operations with specific file size. When they call read/write system call, the write unit, called block size, can be set. In our experiments, we set three types of block size 4KB, 64KB, and 128KB to evaluate the performance for various request size distributions.

We compare the designed method with the EXT2 and JFFS2 original version. The EXT2 is set upon the linux mtdblock ftl layer, and JFFS2 original version means that is constructed without our writeback method. Figure 3 shows the overall experimental results. The latency means the time of read or write for one block. In the Figures, both in EXT2 and JFFS2, the write back method represents very low average write latencies until the file size becomes 16MB. When the file size is over 16MB, average write latencies of them increase dramatically. This threshold value suggests available main memory size for buffering. When the buffers are sufficient to copy users data, these two methods copy file from user address to kernel buffers, but real flushing are not done immediately. However, if buffers are not enough, Linux should flush the dirty buffers immediately to make free buffers. On the other hand, the original JFFS2 shows steady average write

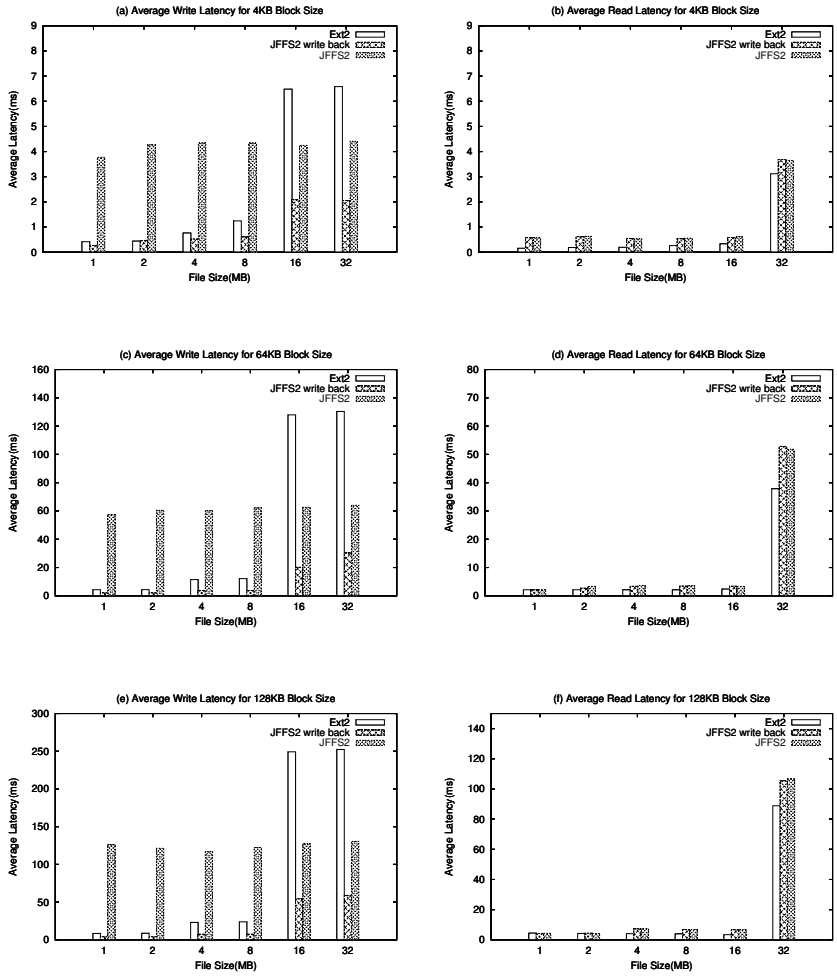

Fig. 3. Experimental Results for Tio Benchmark Program with different file sizes

Filesystem	time / size	<0.4	0.4<	0.5<	1<	10<	100<	10000<
Ext2	1MB	28	211	15				
	2MB	27	443	41				
	4MB	30	899	94				
	8MB	27	1811	197	12			
	16MB	81	3584	405	9	2	14	1
	32MB	89	7315	673	74	7	31	2
	time / size	1.9<x<2	2<	3<	4<	5<	10<	100<
JFFS2	1MB	185	56		13	1		
	2MB	372	110		28	1		
	4MB	726	235	1	56	4	2	
	8MB	1435	403	12	113	44	5	35
	16MB	2791	828	37	261	92	7	79
	32MB	5505	1697	90	488	249	15	147
	time / size	0.2<	0.3<	0.5<	1<	10<	100<	10000<
JFFS2 WB	1MB	243	12					
	2MB	495	15				1	
	4MB	988	32				3	
	8MB	1971	70				6	
	16MB	3926	154	2	3		9	1
	32MB	7786	315	13	55	2	18	2

Table 2. Experimental Results for Tio Benchmark Program, which represents write latency distribution for each 4KB block write

latency while varying the file size, since JFFS2 flushes the requests immediately. The reason that EXT2 and JFFS2 write back has higher latencies than that of JFFS2 when file size becomes 16MB is very large peak of maximum latency for some written blocks, which is due to the OS scheduling effect and CPU utilization of bulk flushing. The flushing overhead is realized from the Table 2, which represents the write latency distribution for 4KB block size, for each filesystem. When system feels lack of free memory, more kernel flush threads occupy cpu utilization to make more free memory by reducing the timer wakeup intervals. Also in wb_flash, as pending requests are increased, number of flushed requests are also increased. Therefore, the maximum latency is thousands of average latency when the file size is large, as shown in Table 2. However, this phenomenon rarely occurs. As block size is increased, experiments give similar results, which increase average latency as block size increases. The read latency similar results of that of write except the average latency of JFFS2 original version. It is from the caching effect of written file.

5 Conclusion

In this paper, we design and implement the efficient I/O mechanism for JFFS2 filesystem based on the flash memory storage. Linux OS generally uses the write back routine for deferred I/O using *pdflush* and *kupdate* kernel thread, which writes back the dirty memory pages and buffers to the real storage medium through the block device layer. However, flash memory based filesystem, JFFS2, does not use block device layer, so they cannot utilize Linux deferred I/O. When

writing new user's data, JFFS2 makes new node which contains both inode information and data, writes to the flash medium using flash driver interface directly. Whenever the write is performed from users, it is not returned from kernel until all the data are written out to flash memory. This write transaction mechanism gives much latency and response time to users and degraded the overall system read/write performance. Therefore, we implement the deferring mechanism of write requests to enhance write latency. For this, we first analyze the write procedure of the JFFS2 filesystem in detail, and derive the drawback and overhead. Then, we design the flash write back routine for deferred I/O whose method is similar to the Linux deferred I/O mechanism. We apply to the JFFS2 flash filesystem by implementing $fflush$ kernel daemon and $flash\_writeback$ routine in Linux kernel. The designed flash write back routine can reduce average latency of write operations when the buffers are enough to get the users data.

References

1. F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers", *In Proc. of the 1st Symposium on Operating Systems Design and Implementation(OSDI)*, 1994, pp 25-37
2. Intel Corporation, "Understanding the flash translation layer(FTL) specification", http://developer.intel.com/.
3. Samsung Electronics Co., "NAND Flash Memory & SmartMedia Data Book", 2002.
4. Samsung Electronics Co., "OneNAND Specification", 2005.
5. "Memory Technology Device (MTD) subsystem for Linux.", http://www.linux-mtd.infradead.org.
6. A. Ban, "Flash file system,", *United States Patent, no. 5,404,485*, April 1995.
7. Card R, Ts'o T, Tweedie S., "Design and Implementation of the Second Extended Filesystem.", *The HyperNews Linux KHG Discussion*, http://www.linuxdoc.org, 1999.
8. A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System", *Usenix Technical Conference*, 1995.
9. D. Woodhouse, "JFFS: The Journalling Flash File System", *Ottawa Linux Symposium*, 2001.
10. Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System". *ACM Transactions on Computer Systems*, 10(1), 1992.
11. Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel",. O'reilly
12. Texas Instruments Co., "OMAP 5912 Startker Kit(OSK)", http://focus.ti.com/omap/docs/omaphomepage.tsp.
13. Threaded I/O bench for Linux http://sourceforge.net/projects/tiobench/