UBI with Logging

Brijesh Singh
Samsung, India
brij.singh@samsung.com

Rohit Vijay Dongre
Samsung, India
rohit.dongre@samsung.com

Abstract

Flash memory is widely adopted as a novel non-volatile storage medium because of its characteristics: fastaccess speed, shock resistance, and low power consumption. UBI - Unsorted Block Images, uses mechanisms like wear leveling and bad block management to overcome flash limitations such as “erase before write”. This simplifies file systems like UBIFS, which depend on UBI for flash management. However, UBI design imposes mount time to scale linearly with respect to flash size. With increasing flash sizes, it is very important to ensure that UBI mount time is not a linear function of flash size. This paper presents the design of UBIL: a UBI layer with logging. UBIL is designed to solve UBI issues namely mount time scalability & efficient user data mapping. UBIL achieves more than 50% mount time reduction for 1GB NAND flash. With optimizations, we expect attach time to reduce up to 70%. The read-write performance of UBIL introduces no degradation; a more elaborate comparison of results and merits of UBIL with respect to UBI are outlined in the conclusion of the paper.

1 Introduction

Flash memories are extensively used in embedded systems for several remarkable characteristics: low power consumption, high performance and vibration tolerance. However flash storage has certain limitations namely “erase before write”, write endurance, bad blocks. The block of a flash memory must be erased before writing again. Besides, each block has limited erase endurance; the block can be erased for a limited number of times. Traditional applications need software assistance to overcome these limitations.

There are two common approaches to deal with the flash limitations. Firstly, a flash translation layer (FTL) that does transparent flash management. It gives a generic disk interface. The traditional file systems like ext2, FAT work unchanged. This approach limits optimizations as file systems are not flash aware.

Second approach uses flash file system. Flash file systems, like JFFS [1], YAFFS [2], are designed to handle flash limitations. In this approach, every flash file system address flash limitations. It is ideal to address them in separate flash layer. This leads us to the third approach. A flash aware file system that can co-operate with a software layer for optimum flash usage. UBI [3] is a software layer designed to follow this approach.

UBI is a flash management layer which also provides volume management. A UBI volume can be a static volume or a dynamic volume. For flash management, UBI provides following functionalities.

- Bad block management
- Wear leveling across device
- Logical to Physical block mapping
- Volume information storage
- Device information

2 Related Work

UBI was developed in 2007. UBI gives logical block interface to the user; each logical erase block (LEB) is internally associated with a physical erase block (PEB). This association is called “Erase Block Association (EBA)”. EBA information of each PEB is stored in VID header. VID header of a physical block resides in the same block. Apart from this, UBI also stores EC header in each physical block; EC header stores erase count of the block. Typical UBI block structure is shown in Figure 1. Initialization of UBI demands processing of both headers from every block. UBI scans complete flash in order to build in-RAM block associations. This introduces a scalability problem. UBI's initialization time scales linearly with respect to flash size; increase in flash size increases mount time of UBI. With flash sizes increasing up to several GB's, it is very important to ensure that UBI mount time is not a linear function of flash size.

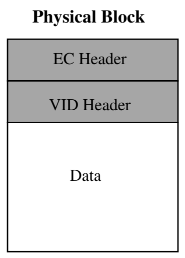

Figure 1: UBI Physical Block Structure

Lei et al. [4] proposed Journal-based Block Images (JBI) which focuses on reducing number of write operations and flash space requirement. To achieve this, JBI uses fragmented mapping table and journal system. Limited work has been carried out to reduce mount time of UBI. To address mount time scalability issue, it is important to avoid scan of complete flash. Possible solution to this problem is to store mapping information in fixed group of blocks on flash.

3 UBIL: UBI with Log

In this paper we present UBIL: "UBI with Log", to address mount time scalability issue. In order to reduce initialization time, UBIL stores block mapping information to the flash. This design consists of super block, commit block and EBA log. Super block which stores location information for commit block and EBA Log, is stored at fixed physical location. Commit block is a snapshot of valid UBI block mapping. EBA Log is a difference between present state and last commit. Commit and EBA Log can move anywhere in flash. Hence these blocks are wear-leveled.

3.1 Super Block (SB)

Super block is stored at two erase blocks in flash. First super block instance is present in first good erase block and second instance is present in last good erase block. The two instances of super block are not mirror of each other. Instead, only one of them contains valid super block entry. Every super block entry occupies page size of flash. To update super block, instead of erasing and writing the block, we log the super block. It means, any update to super block is written in one of the physical blocks.

Super block is written alternatively to one of the two copies (like ping-pong table). As shown in figure, first super block entry 'Entry0' is written on first block, SB0. Next entry 'Entry1' is written to second super block SB1. Subsequent entries Entry2, Entry3... are written alternatively in each block. This gives advantage over mirroring as space is not wasted. Also this improves lifetime of physical blocks reserved for super block.

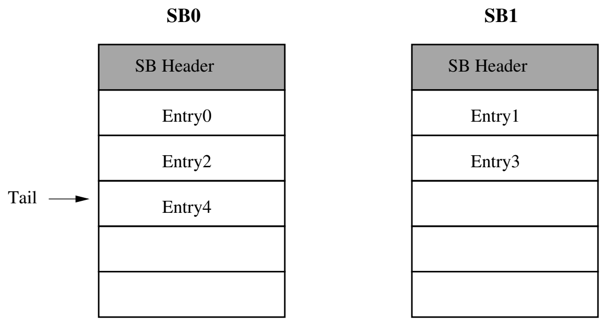

Figure 2: Super Block Update Sequence

While reading super block, we scan through super blocks and find latest written entry, which is a valid super block entry. In Figure 2, valid super block entry is pointed as tails. Writing super block entry may fail. In such situations, other instance of super block contains valid entry.

3.2 Commit block (CMT)

Commit contains mapping information. Size of commit is decided at the time of Ubinize. Depending on partition size, commit may span up to multiple PEBs. Commit information is crucial. Hence two mirror copies of commit are maintained. Even if one of the copies is correct, it is possible to recover the commit. For clean detach, UBI uses commit information during subsequent attach. In case of failure replay of EL is done to restore latest state. Super block contains two map information of commit; present commit and future commit. During commit process, list of future commit blocks in super block is updated first. Then commit is written to these blocks. On successful completion, super block is updated replacing present commit by new commit. Hence commit operation is atomic and tolerant to power failure. If commit is incomplete during detach, all the failed

commit blocks are recovered and given for garbage collection. EL becomes invalid after commit. New empty log is initialized during commit.
Note: UBIL gives option of compressing CMT. This decreases average read/write time of CMT.

3.3 EBA Log (EL)

EBA log contains mapping information of each physical erase block updated after last commit. Hence EL is difference between last commit and present UBI state. Each EL entry contains "EC and VID header" of a physical erase block. EL may contain valid and invalid entries. When EL gets full, only valid entries are written to the commit. After successful commit, old EL is invalid and fresh log is created. This operation is done by reserving new PEBs for EL and handing over old PEBs for garbage collection.
Note: It is possible to configure number of blocks allocated to EL at compile time.

4 UBIL: Initialization

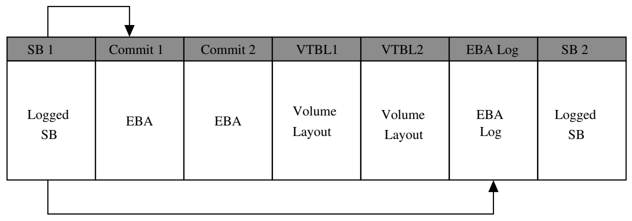

Figure 3: UBIL Flash Layout

UBIL flash layout is shown in Figure 3. UBIL initialization starts with reading super block and finding latest super block entry. Super block locates CMT and EL. For good detach, initialization involves reading CMT. For bad detach, some of mapping information may be present in EL. Hence, initialization involves reading CMT and replaying EL. After successfully reading CMT and EL, other sub-systems of UBIL are initialized. This includes volume initialization, wear-leveling initialization and EBA initialization. During initialization if one of CMT, SB or EL shows recoverable read errors, UBIL initialization proceeds. In this case, after successful initialization of all sub-systems, commit process is called. This guarantees that, CMT is moved to safer erase block, less vulnerable for corruption. Due to removal of scanning, UBIL initialization time is very less as compared to UBI. Steps followed in UBIL initialization are outlined below.

1. Find latest super block by finding tail of super block.
   (a) If the tail is bad (power cut happened while writing super block) the other super block PEB contains valid super block entry.
2. Locate CMT, EL blocks from super block.
3. Generate latest snapshot of UBI.
   (a) Read CMT.
   (b) Apply EBA Log.
4. If previous commit has failed, recover reserved blocks for commit.
5. Initialize Volumes.
6. Initialize Wear leveling.
7. Initialize EBA information.

5 Performance Measurement

We have compared performance of UBIL against UBI on SLC NAND flash. Mount time performance and read-write performance tests were conducted. Tests were performed on Apollon board with OMAP 2420 chipset having 64 MB RAM. We tested UBIL with Linux kernel 2.6.33.

5.1 Mount time performance

UBI attach time increases linearly to partition size. This is due to scanning of complete flash. In case of UBIL, commit size increases with increase in flash size. Causing UBIL attach time to increase marginally. But this increase is very minimal in comparison to UBI. Mount time performance comparison is shown in Figure 4. It is evident that UBIL performs far better than UBI. As partition size increases, UBIL performs better than UBI in terms of attach time. UBIL achieves more than 50% attach time reduction for 1 GB NAND flash.

As partition size increases, UBIL performance better than UBI in terms of attach time. UBIL achieves more than 50% attach time reduction for 1GB NAND flash.

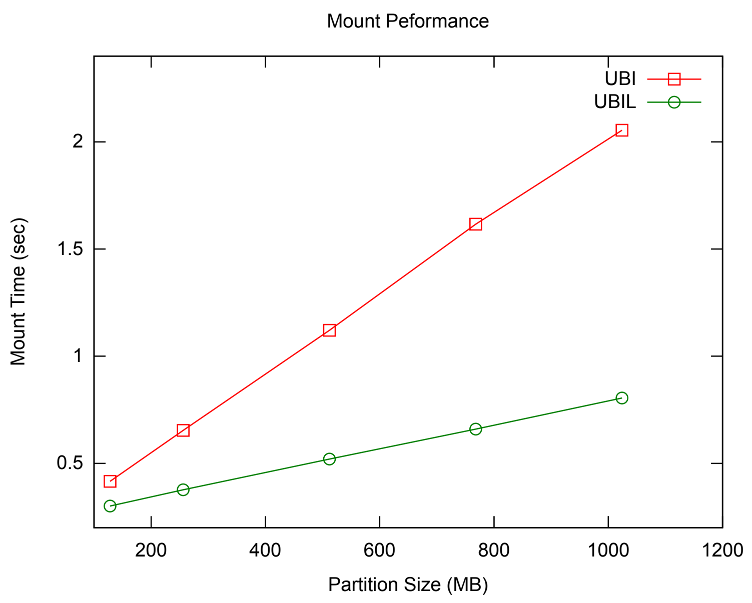

Figure 4: Mount Performance : UBIL vs UBI

5.2 Read-Write Performance

This test measures actual file system read-write performance. For performing this test we used Iozone running on partition mounted with UBIFS. In read-write test we performed sequential and random read-write tests. Performance measurements are given in Table 5.2. It can be inferred from table that there is no significant effect on read-write performance. This is because, UBIL EL writing frequency is same as meta data write frequency of UBI.

Table 1: IO Performance

Operation	UBI (MB/s)	UBIL (MB/s)
Read	6.33	6.33
Write	3.49	3.71
Re-read	6.33	6.33
Re-write	3.39	3.64

6 Conclusion and Future Work

In this paper we presented UBIL to effectively deal with mount time scalability issue of UBI. While UBI stores mapping information across flash, we maintained mapping information at one place. This significantly reduce mount time by avoiding full flash scan. Bedsides UBIL, do not perform any extra read-write operation, causing read-write performance comparable to UBI. As discussed in results, Our approach reduces mount time by 50% without affecting read-write performance.

Commit process can be optimized in future by writing EBA mappings directly to the flash. As per present UBIL design, super block is written at fixed location. These blocks are not wear-leveled. Super block handling can be improved by using block chaining scheme as discussed in JFFS3 [6] design.

References

[1] D. Woodhouse, *JFFS: The Journaling Flash File System*, In Proceedings of 2001 Linux Symposium, Ottawa, Canada, July 25-28, 2001

[2] *YAFFS: Yet Another Flash File System*, http://www.yaffs.net

[3] T. Gleixner, F. Haverkamp, A. Bityutskiy, *UBI-Unsorted Block Images*, http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf

[4] Lei Jiao, Y. Zhang, W. Lin*Journal-based Block Images for Flash Memory Storage Systems*, The 9th International Conference for Young Computer Scientists, pp. 1331-1336, 2008

[5] D. Woodhouse, *Memory Technology Device (MTD) Subsystem for Linux*, http://www.linux-mtd.infradead.org/doc/general.html, Feb 2010

[6] A. Bityutskiy, *JFFS3 Design Issues*, Version 0.25, http://www.linux-mtd.infradead.org/doc/JFFS3design.pdf, Oct 2005

[7] Brijesh Singh, Rohit Dongre, *UBIL Performance Log for NAND*, http://git.infradead.org/users/brijesh/ubil_results /blob/HEAD:/nand_mount_ti me.pdf

[8] Brijesh Singh, Rohit Dongre, *UBIL- UBI with Log*, Source code, http://git.infradead.org/users/brijesh/ubi-2.6.git

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada