Design Document

Unsorted Block Images Enhancement

For Linux 2.6

Primary Author(s): Brijesh Singh

Second Author(s): Rohit Dongre

Revision 1.01

First Updated: 02/3/2009 - 1:32 PM

Last Updated: 08/05/2010 - 11.40 PM

Note: This document requires prior knowledge of UBI

Abstract

Current UBI design scatters metadata through-out the flash. Although this design suites the flash property, it enforces a scalability issues. The initialization time of UBI increases linearly with the size of Flash. This is due to metadata location of current UBI.

This paper presents design change in UBI: a UBI with logging. The design focuses on reducing initialization time of UBI. The read-write performance of UBIL is similar to UBI. UBI features like power consistency, bad block management are maintained in UBIL. A more elaborate design description is outlined in the paper.

Table of Contents

Unsorted Block Images Enhancement 1

Abstract 2

1. Background 4

1.1. UBI Flash layout 4

1.2. UBI volume layout 5

1.3. Requirement for new design 5

1.4. Design Goals 5

2. UBI with Log 6

2.1. Flash Layout 6

2.2. Super Block (SB) 6

2.3. Commit (CMT) 6

2.4. EBA Log (EL) 7

2.5. PEB INFO 7

2.6. UBINISING 8

2.7. Initialization 8

3. On Flash Headers 8

4. UBIL Features 9

5. Suggestions 9

6. Glossary 9

7. Bibliography 9

1. Background

UBI1 stands for "Unsorted Block Images". UBI is volume management system for raw flash devices; it manages multiple logical volumes on a single physical flash device. UBI spreads the I/O load across the flash device to perform wear-levelling. Apart from wear-levelling UBI jobs include bad block management, bit flip handling, out of place updates (atomic update) and tolerance to power failures or unclean reboots.

1.1. UBI Flash layout

At any instance, a physical block is associated to maximum one logical block. This association is called erase block association (EBA). EBA information of each physical erase block is stored in the same physical erase block. This header is called Vid[2] header. UBI also stores EC[3] header in the physical block; EC header is erase count information of the current block. UBI scans the flash during initialization to create an in ram map of the flash. This introduces a problem. UBI initialization time increases linearly with flash size; as flash size increases, initialization time increases.

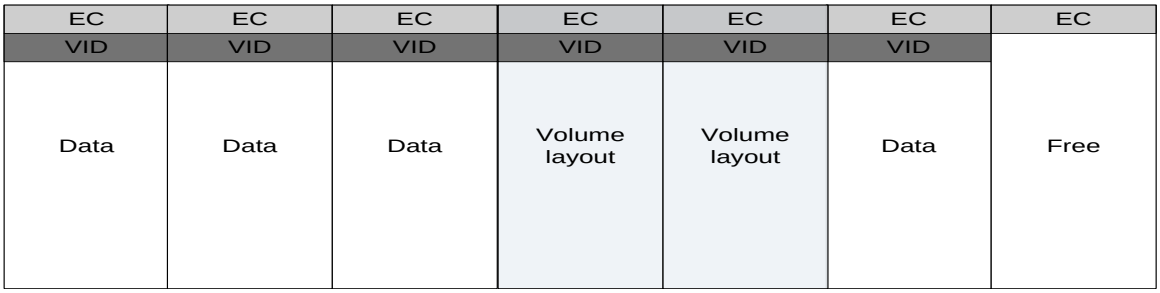


Figure 1 UBI flash layout

EBA association can change due to following reasons

- New block association
- Out of place updates (atomic update)
- Block movement done by Wear-levelling
- Bad block handling
- Bit-flip handling.

[1] http://www.linux-mtd.infradead.org/doc/ubi.html
[2] http://www.linux-mtd.infradead.org/doc/ubi.html#L_ubi_headers
[3] http://www.linux-mtd.infradead.org/doc/ubi.html#L_ubi_headers

1.2.UBI volume layout

Volume layout information is stored in two physical erase blocks. These blocks are mirror copy of each other. The Vid information of these blocks point to the layout volume. These blocks are located in Initial scan. UBI then creates user interfaces for these volumes.

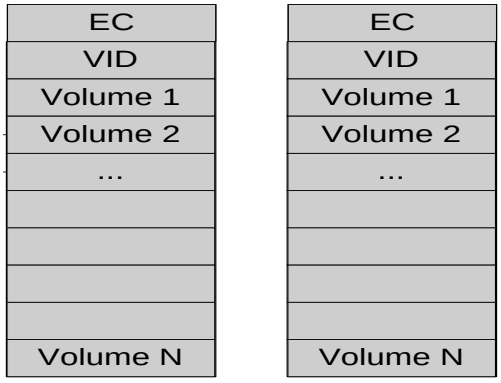


Figure 2 Volume layout of UBI

1.3.Requirement for new design

To reduce initialization time, it is ideal to modify or remove scanning process. This is possible by introduction of commit blocks. All the erase block association information can be kept in the commit blocks. These blocks can be read for faster initialization. However, flash property avoids from over-writing block without erasing. It is imperative that these blocks should be wear-levelled.

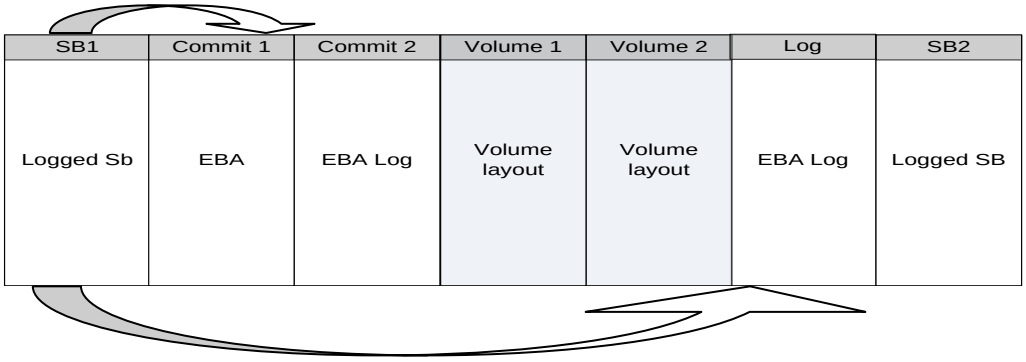
1.4.Design Goals

- To minimize the initialization time of UBI
- To maintain current functionalities
- To maintain tolerance to power cuts
- To keep all metadata together in one / multiple blocks
- To make UBI scalable in terms of mount time
- Minimal code change.

2. UBI with Log

2.1.Flash Layout

Figure 3 Flash layout of UBIL



New design consists of super block, commit and Log. Super block is the only metadata information in UBI which is at fixed physical erase block. Super block locates commit and Log. Commit is snapshot of valid UBI logs. UBI Log is difference between commit and present state of UBI.

2.2.Super Block (SB)

Super block contains location information for EBA log and commit block. Two copies of super block are maintained; first copy is present in first good erase block, second copy is present in last good erase block. Super block occupies one page of flash. To update super block, instead of erasing and writing the entire erase block, we log the super block. It means, any update to super block will be written in same physical erase block towards the tail. While reading, tail of this log is considered as the valid super block.

The two copy of super block are not mirror of each other. Instead only one of them will contain valid super block. Super block is written alternatively to one of the two copies (like ping-pong table). This gives advantage over mirroring as space is not wasted. Super block may go bad during write; in this case the other copy contains the valid entry.

2.3.Commit (CMT)

Commit contains PEB 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 or anchor block contains two map information of commit; present commit and future commit. During commit process, future commit blocks in super block are 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 log becomes invalid after commit. New empty log is initialized during commit.

2.4.EBA Log (EL)

EBA log(EL) 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 to gc.
Note: It is possible to configure number of blocks allocated to EL at compile time.

2.5.PEB INFO

PEB info header stores information of a physical erase block. PEB info is present in CMT and optionally in EL.
The PEB INFO structure is described as follows.

```
struct peb_info {
        __u8 status;
        struct ubi_vid_hdr v;
        __be32 ec;
} __attribute__((packed));
```

The one byte status of PEB tells which list this peb belongs. The status can also be used for marking a PEB bad. In future, status can be used to avoid bad block scanning by mtd drivers. Mtd drivers can form bbt table on demand4. UBIL shares the same Vid header as that of UBI. It is possible to reduce un-necessary information from Vid header. EC is erase count of current block.

[4] BBT management in NAND is usually done by scanning oob status information.

2.6.UBINISING

Ubinising is the process of formatting flash with UBI image. This process follows following steps.

1. Check for sufficient space in flash
2. Find first and last good block for super block
3. Erase the flash
4. Build in ram PEB info table.
5. Write default volume layout
6. Write commit block
1. Write super block

Step 3 is deliberately done; this gives clean in ram PEB INFO map. If all erase blocks are not erased in this process, gc will erase them on demand. Each erase causes one Log entry to be written.

2.7.Initialization

- Locate erase blocks containing super block
- Find latest super block by finding tail of super block PEBs
    - If the tail is bad (power cut happened while writing super block) the other super block PEB contains latest super block
- Locate CMT, EL blocks from super block
- If CMT has failed, recover last state
- Else read CMT
- Generate latest snapshot of UBI
    - CMT is the read into peb_lookup buffer
    - Log is applied to this buffer
- Create EBA information
- Initialize Volumes

3. On Flash Headers

```
struct ubi_sb {
        struct node_t lh;
        __be32 version;
        __be32 cmt_status;
        __be32 el_resrvd_buds;
        __be32 cmt_resrvd_buds;
        __be32 cmt_next_buds;
        __be32 vtbl_peb [2];
        __be32 buds [0];
} __attribute__ ((packed));
```

Note: vtbl peb is location of volume table layout. This is not used in current code. In future, this can be used to locate volume table without EBA table being present.

```
struct el_node {
        struct node_t lh;
        struct peb_info recs [0];
} __attribute__((packed));
```

El node is group of peb information. PEB's are grouped according to their offset.[5]

4. UBIL Features

- Faster initialization.6
- Power cut tolerance similar to UBI; this is because frequency of writing Log is same as updating EC or VID header.
- No change in Atomic update.
- Read/ Write speed almost same as UBI

5. Suggestions

- Instead of writing valid log entries to commit, it is possible to write in ram EBA map in commit. This will give optimum results in terms of initialization time.
- Multithreaded log, with GC support. This may be used to exploit parallel writes in flash.

6. Glossary

- UBI : Unsorted Block Images
- PEB : Physical Erase Block
- LEB : Logical erase Block
- EC header : Erase Counter Header(UBI 1.0)
- VID header : Volume Identification Header(UBI 1.0)
- EL: EBA Log
- CMT: Commit
- SB: super block or anchor block

7. Bibliography

1. This site has all the information for anything related to Memory technology devices,
   - http://www.linux-mtd.infradead.org/
2. UBI design.
   - http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf
3. Similar problem: JFFS3 Design
   - http://www.linux-mtd.infradead.org/doc/JFFS3design.pdf
4. JFFS2 Documentation.
   - http://sources.redhat.com/jffs2/jffs2.pdf
5. Performance Log for NAND:
   - http://git.infradead.org/users/brijesh/ubil_results/blob/HEAD:/nand_mount_time.pdf

5 http://lists.infradead.org/pipermail/linux-mtd/2009-February/024483.html

6 http://git.infradead.org/users/brijesh/ubil_results/blob/HEAD:/nand_mount_time.pdf