A Stackable Wear-Leveling Module for Linux-Based Flash File Systems

Hai-Ning Wu[1] and Li-Pin Chang
hainingwu@gmail.com and lpchang@cs.nctu.edu.tw
Department of Computer Science
National Chiao-Tung University, Hsin-Chu, Taiwan

Abstract
Flash memory has now become a crucial component in building Linux-based embedded computers. As the overall flash-memory lifetime is concerned with block endurance, wear leveling is needed to evenly erase all blocks. This paper presents a modularized implementation of a wear-leveling algorithm. Our goal is to instantly enable any existing Linux-based flash file systems the ability of wear leveling without to modify the file systems.

1. Introduction

With rich resources and trustworthy reliability, Linux has been a promising choice in building advanced embedded computers. A commonly faced design problem is how a storage system can be deployed in embedded computers, as power-hungry hard drives are vulnerable to shock. As a result, flash memory is widely used. Different from disks, flash memory is write-once and bulk-erase.

New flash file systems are introduced to Linux, such as JFFS2 [1] and YAFFS [2]. Basically, a flash file system must deal with *address translation* and *garbage collection*. The former issue is to map the logical addresses of a piece of data onto physical locations on flash memory, and the latter one is to recycle space occupied by invalid data by means of block erasure. In Linux, the MTD (memory technology device) subsystem is introduced as a software framework for solid-state devices, especially for flash memory, as shown in Figure 1.

As realistic workloads access flash memory with strong spatial localities, frequently updated data quickly leave invalid data in some particular blocks. Erasure is thus directed to the blocks in favor of garbage-collection efficiency. However, the preference may ultimately lead to uneven wearing of blocks, as shown in Figure 2(a). Because each individual flash-memory block endures only a limited cycle number of erasure operations (typically 100K) [3], the overall lifetime of the storage systems is largely concerned. Wear leveling refers to the intention to evenly distribute erasure cycles over blocks, as shown in Figure 2(b).

[1] The corresponding author.

The current open-source flash file systems do not seriously consider wear leveling. In particular, JFFS2 periodically erases an infrequently worn block

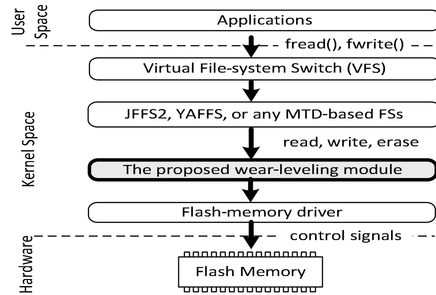

Figure 1. The architecture of the Linux MTD subsystem and the role of the proposed wear-leveling module.

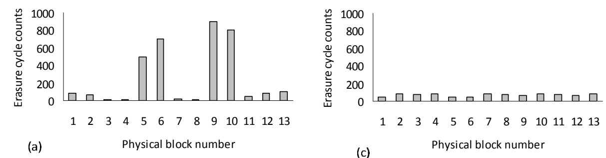

Figure 2. Different effects on wear leveling: (a) No wear leveling, and (b) the desired effect.

, and YAFFS considers no wear leveling at all. One brute-force approach is to directly modify the file systems. However, obviously, different file systems need to be modified separately, and the wear-leveling code must be synchronized with version changes to the file systems. In this paper, we present a stackable wear-leveling module. As shown in Figure 1, the module sits between real flash-memory devices and file systems. Wear-leveling activities are conducted internally in the wear-leveling module, and they are completely transparent to file systems. By this way any Linux-based flash file systems can be benefited without any modifications.

2. A Stackable Wear-Leveling Module

There are two kinds of wear-leveling algorithms in literatures. Reclaiming-based algorithms are garbage-collection policies that take wear leveling into considerations. Placement-based algorithms do nothing but change data placement. It is to manipulate that which block is erased by garbage collection, and can be orthogonal to garbage collection.

We have previously proposed an efficient and

effective placement-based wear-leveling algorithm, the *dual-pool algorithm* [3]. The two key ideas behind its design are: (1) *cold data migration*; to cease the wearing of old blocks (those have been badly worn) by placing cold data (infrequently updated data) in the blocks, and (2) *hot-cold regulation*; to develop the effects of wear-leveling activities by protecting the blocks against being repeatedly involved. Detailed algorithm descriptions can be found in [3].

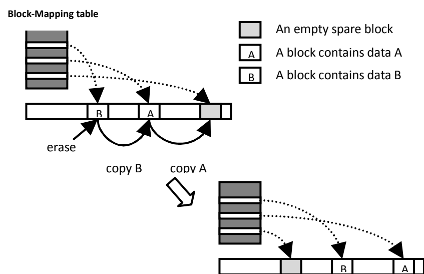

Figure 3. How the wear-leveling module conducts data migration and hides it from the upper-level file systems.

Our design issue is how the changes of data placement made by the dual-pool algorithm can be transparent to file systems. For this purpose, a pseudo MTD device driver is created. The pseudo driver sits between real MTD devices and file systems. The pseudo driver reacts to the upper-level file system as if it were one ordinary MTD device. It accepts reads, writes, and erasures, and then passes them to the underlying real MTD device(s).

The wearing of physical flash-memory blocks are monitored by the pseudo MTD device. Whenever the wearing of all the blocks is becoming uneven, the dual-pool algorithm may decide to change the placement of data. Three blocks would be involved in this procedure. As shown in Figure 3, let the blocks storing data A, B and the spare block be referred to as Ba, Bb, and Bs, respectively. The first step is to copy data A from Ba to Bs. After this, Ba can be erased, and data B are copied from Bb to Ba. Finally, Bb is erased and becomes a new spare block.

However, the upper-level file system must not see the changes of data placement. To achieve this, the pseudo MTD device uses a RAM-resident block-mapping table. The table translates the block addresses of the pseudo MTD device to the block addresses of the underlying (real) MTD device. As shown in the second step of Figure 3, the mapping is revised so that data A and data B are always referred to by the same block addresses of the pseudo MTD device.

3. Experimental Results

We have conducted experiments on our implementation of the wear-leveling module. The underlying MTD device is a 4MB MTD device, of which the block size is 4KB and therefore there are 1024 blocks. The pseudo MTD device is stacked on the 4MB MTD device, and JFFS2 mounts over the pseudo MTD device. In the file system, a 3200KB read-only file is created as cold data. A program repeatedly re-writes a 400K file to simulate the behavior of hot data. Note that the aggressiveness of the dual-pool algorithm is configured by parameter **TH**. The smaller the TH is, the more aggressive the wear-leveling activities would be.

Figure 4 shows the distributions of block-erasure cycles by the end of experiments. The wearing of blocks is quite uneven when JFFS2 is solely used. By inserting our wear-leveling module, the block-erasure cycles are significantly leveled. Figure 5 shows the standard deviations and averages of all the block-erasure cycles. It shows that wear leveling is achieved without wasting many precious erasure cycles.

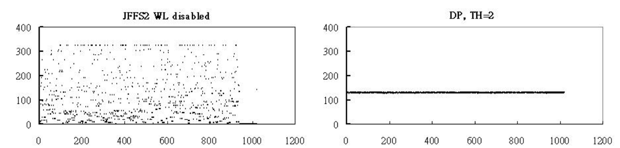

Figure 4. Distributions of erasure cycles of (a) JFFS2 and (b) JFFS2 with the wear-leveling module (TH=2). X axes and Y axes represent block addresses and erasure cycles, respectively.

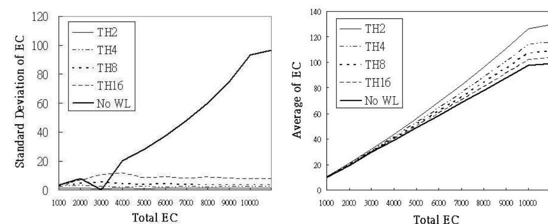

Figure 5. The standard deviations and the averages of all block-erasure cycles. The smaller the numbers the better.

4. Conclusion

This paper considers an approach that enables any Linux-based flash file systems the ability of wear leveling. Instead to individually modify each file system, a stackable wear-leveling module is introduced. As the scale of embedded software grows rapidly, such a modularized design is a graceful approach to ratability, versatility, and scalability. The source code of the wear-leveling module is available at http://esslab.tw/wl_mod.zip

References

[1] D. Woodhouse, ``JFFS: The Journaling Flash File System," Proceedings of Ottawa Linux Symposium, 2001.

[2] C. Manning and Wookey, ``YAFFS Specification," Aleph One Limited,, Dec, 2001.

[3] L. P. Chang, ``On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems," Proceedings of the 22nd ACM Symposium on Applied Computing, 2007.